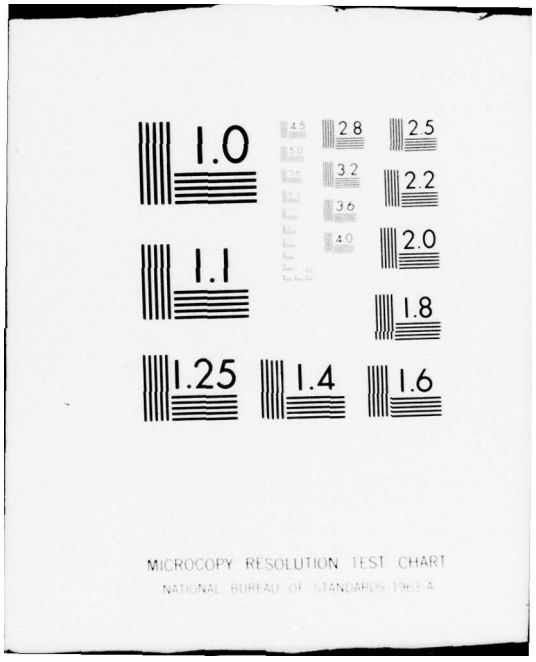
AD-A057 802 ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MASS C--ETC F/G 9/2
AN APPROACH TO CONSTRUCTING FUNCTIONAL REQUIREMENT STATEMENTS F--ETC(U)
JUN 78 S L HUFF, S E MADNICK N00039-78-G-0160
UNCLASSIFIED CISR-P010-7806-06 NL

|OF|
AD
A057802

END
DATE
FILMED
10-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

(12)

LEVEL II

Center for Information Systems Research
Massachusetts Institute of Technology
Alfred P. Sloan School of Management
50 Memorial Drive
Cambridge, Massachusetts, 02139
617 253-1000

78 08 18 009

Contract Number N00039-78-G-0160

Internal Report Number P010-7806-06

Deliverable Number 002

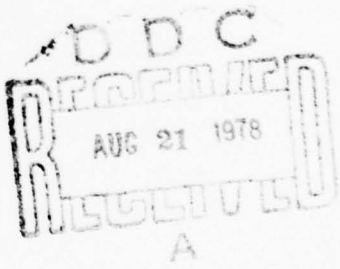
An Approach to Constructing Functional Requirement Statements for System Architectural Design

Technical Report #6

S. L. Huff

S. E. Madnick

June 1978

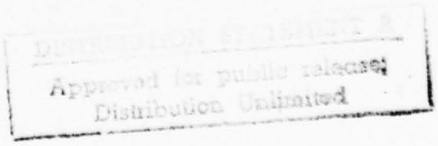


Principal Investigator:

Prof. S. E. Madnick

Prepared for:

Naval Electronic Systems Command
Washington, D.C.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Technical Report #6

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*: "An Approach to Constructing Functional Requirement Statements for System Architectural Design"

5. TYPE OF REPORT & PERIOD COVERED: Technical rept.

6. PERFORMING ORG. REPORT NUMBER: CISR-P010-7806-06, CISR-TR-6

7. AUTHOR(s): S. L. Huff, S. E. Madnick

8. CONTRACT OR GRANT NUMBER(s): N00039-78-G-0160

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Center for Information Systems Research, Sloan School of Management, M.I.T., Cambridge, Mass. 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Electronic Systems Command

12. REPORT DATE: June 1978

13. NUMBER OF PAGES: 12 90p.

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*: Unclassified

15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release. Distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

System requirements analysis; functional requirements specification; software architectural design; problem design structuring.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The objective of this study is to develop a systematic approach to the architectural design of complex software systems. This report builds on earlier work which has proposed and tested a decomposition technique for generating an architectural design structure out of a set of functional requirement statements and their interrelationships. In this report, the problem of creating functional requirements, which become the "input" to the design structuring methodology, is investigated. Presently ~~over~~

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

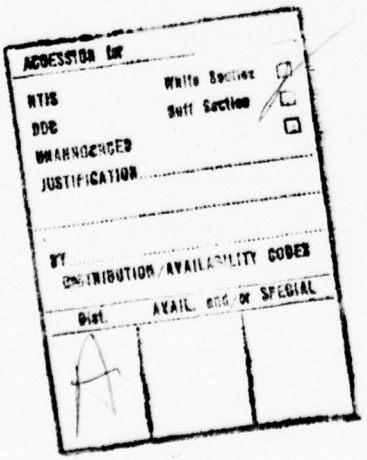
S/N 0102-014-6601

409 790

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

next page

available "requirement statement languages", such as PSL ("Problem Statement Language") are examined, and the conclusion is drawn that such schemes are not very appropriate for the expression of requirements prior to system design. Certain key concepts used frequently in the requirements analysis and specification literature are then examined and clarified, and related together in the context of a simple framework. An alternative approach to specifying requirements, based upon a set of "requirement templates", is proposed and its application illustrated. Finally, the appendices contain detailed information regarding PSL syntax and semantics, as well as an application of the template approach to the requirements for a real-life database management system.



PREFACE

The Center for Information Systems Research (CISR) is a research center of the M.I.T. Sloan School of Management. It consists of a group of management information systems specialists, including faculty members, full-time research staff, and student research assistants. The Center's general research thrust is to devise better means for designing, implementing, and maintaining application software, information systems, and decision support systems.

Within the context of the research effort sponsored by the Naval Electronics Systems Command under contract N00039-78-G-0160, CISR has proposed to conduct basic research on a systematic approach to the early phases of complex systems design. The main goal of this work is the development of a well-defined methodology to fill the gap between system requirements specification and detailed system design.

The research being performed under this contract builds directly upon results stemming from previous research carried out under contract N00039-77-C-0255. The main results of that work include a basic scheme for modelling a set of design problem requirements, techniques for decomposing the requirements set to form a design structure, and guidelines for using the methodology developed from experience gained in testing it on a specific, realistic design problem.

The present study aims to extend and enhance the previous work, primarily through efforts in the following areas:

1) additional testing of both the basic methodology, and proposed extensions, through application to other realistic design problems;

2) investigation of alternative methods for effectively coupling this methodology together with the preceding and following activities in the systems analysis and design cycle;

3) extensions of the earlier representational scheme to allow modelling of additional design-relevant information;

4) development of appropriate graph decomposition techniques and software support tools for testing out the proposed extensions.

This document, which relates primarily to category (2) above, includes an analysis of the general problem of stating functional requirements for a desired system, an investigation of the applicability of certain presently available techniques for stating system requirements, and a proposed new approach for requirements specification that could be used effectively within the overall design methodology emerging from this research.

EXECUTIVE SUMMARY

Complex design problems are characterized by a multitude of competing requirements. System designers frequently find the scope of the problem beyond their conceptual abilities, and attempt to cope with this difficulty by decomposing the original design problem into smaller, more manageable sub-problems. Functional requirements form a key interface between the users of a system and its designers. In this research effort, a systematic approach has been proposed for the decomposition of the overall set of functional requirements into sub-problems to form a design structure that will exhibit the key characteristics of good design: strong coupling within sub-problems, and weak coupling between them.

In this report, the problem of creating the functional requirements, which become the "input" to the design structuring methodology, is investigated. Presently available "requirements specification languages", such as PSL/PSA, are examined, and the conclusion is drawn that such schemes are not very appropriate for the expression of system requirements prior to system design.

Then, certain key concepts and terminology used frequently in the requirements analysis and specification literature are examined and clarified, and related together in the context of a simple framework. This framework illustrates the transition of system requirements from capability-oriented to process-oriented specifications, and the changes - toward lower levels of abstraction, and toward greater procedurality - that such requirements undergo as system design and development is acrried out. The framework provides a way of relating various design schemes to each other and to the overall system life cycle.

In the final section, an alternative approach to specifying requirements, based upon a set of six "requirement templates", is proposed and illustrated. The template approach stresses the need to maintain a high level of semantic flexibility in the early functional specifications, while simultaneously structuring the individual requirements for effective execution of the graph modelling and decomposition steps that occur later in the design methodology. An example application of the template approach is also given.

TABLE OF CONTENTS

1. Introduction. 6

2. The SDM Architectural Design Methodology. 9

2.1 The System Development Life Cycle. 9

2.2 Application of Computers in Functional Development. 12

2.3 The SDM Approach. 13

3. Requirements Statement Languages - The Case of PSL. 17

3.1 The Structure of PSL. 17

3.2 Using Objects and Relationships to Create PSL Statements. 22

3.3 An Illustration of PSL - Part A. 25

3.4 An Illustration of PSL - Part B. 27

3.5 PSL As a Design Tool. 31

4. Terminology and Concepts - Some Clarifications. 34

4.1 Levels of Procedurality. 34

4.2 Types of Requirements. 37

4.3 Processes and Capabilities. 39

4.4 A Framework for Requirement Statements. 41

5. Expressing Functional Requirements. 48

5.1 The Format of Typical Functional Requirements. 49

5.2 Requirement Statement Templates. 51

5.3 Side Effects of Expressing Requirements in Template Form. 58

5.4 Summary. 62

6. Summary and Directions for Further Work. 63

REFERENCES. 69

APPENDIX A. Complete Listing of PSL Statement Types. 72

APPENDIX B. The Full Set of DBMS REquirements. 77

APPENDIX C. Edited DBMS Requirements Transformed to Template Form. 82

1. Introduction.

The problem of designing quality software systems has existed practically as long as computers themselves. In recent years, continued growth in the size and complexity of such systems, and changing economics of computer use, has caused the software design problem to become especially important. As the emphasis in software development continues to shift toward maintainability, reliability, and "understandability", the importance of good design will become ever more acute (Boehm 1973).

The developing body of new theories and techniques pertaining to various aspects of system design and development is usually referred to collectively as "software engineering." Software engineering has its roots in the programming area (Buston, 1969), and much of the research in this field today is still concerned with writing better computer programs. Other tasks within the computer system design and development (CSDD) cycle, being less structured than programming and consequently less amenable to "hard" analysis, have received considerably less attention in software engineering research (Davis 1977). One such task, the one with which we are primarily concerned here, is preliminary (or "architectural") design of software systems.

The Center for Information Systems Research (CISR), at the Sloan School of Management, M.I.T., under contract with the U.S. Navy Electronic Systems Command, has been investigating the architectural design issue, and has developed a new methodology for performing this activity. This methodology- termed "A

Systematic Design Methodology", or "SDM"- is described briefly later in this report, and in more detail elsewhere (see earlier technical reports by Andreu and Madnick, cited in the references).

The SDM approach to system design begins with a description of the functions to be performed by the target system, expressed as a set of individual functional requirement statements. The focus of this report is the creation of these functional requirements specifications for a target system, statements which become the "input" to the SDM design process.

We begin the next section with a relatively brief overview of the SDM methodology, under development in the course of this research.

In Section 3, we investigate the possibility of adopting one of the currently available "requirement statement languages" for the purpose of expressing functional requirements for use in design structuring. The particular scheme evaluated in detail here is the PSL/PSA system, developed as part of the ISDOS project at the University of Michigan (Teichroew 1977). The general structure of PSL, the various PSL statement types, and a two-part illustration of its use are presented.

With the concepts and techniques underlying both SDM and PSL as background, certain terminological and "philosophical" issues regarding requirements specification are addressed in Section 4. One of the special difficulties underlying research in this area is that researchers often do not agree on the meaning of the terms they use. To a casual reader, many of these research

efforts may sound as though they are addressing the same problem, when in fact they may be focused on quite different things. The analysis in Section 4 leads to a recognition of what issues some of these current research efforts are <u>really</u> addressing, and what problem areas remain significantly under-researched.

A key conclusion of Section 4 is that the available requirement statement languages such as PSL, for various reasons, are not really suitable for <u>initially</u> formulating a system's functional specifications. Rather, they are generally better suited for use at a later stage in the design cycle.

In Section 5 we present an approach for structuring English prose-style statements of functional requirements into a form appropriate for use with the architectural design methodology previously developed. This approach, based on a set of six "templates", or general statement patterns, stresses the need to maintain a high level of semantic flexibility in the functional specification, while at the same time structuring the individual requirements for graph modelling and decomposition analysis that forms the methodology. An example application of the template approach is also given.

Section 6 summarizes this report.

2. The SDM Architectural Design Methodology

In order to better place the later sections of this report in context, this section reviews both the motivation behind, as well as the nature of, the system design concepts and methodology being developed in the course of this research project.

2.1 The System Development Life Cycle.

System architectural design is fruitfully viewed as one stage in the overall system development life cycle, as illustrated in Figure 2.1. There, the life cycle for a typical CSDD project is shown to consist of three broad phases, each phase being broken down into more narrowly defined stages. The first, or functional, phase is concerned with:

(a) determining what the computer system is intended to do, in the eyes of the user (stage 1);

(b) translating the user's functional requirements into system requirements (stage 2);

(c) drawing up a high-level preliminary design for the target system (stage 3).

The second, or procedural, phase involves:

(a) generating a detailed design for the target system, usually including program module definitions, interface specifications, etc. (stage 4);

(b) writing the computer programs, in a suitable programming language, to accomplish the function of each module (stage 5);

(c) making sure that the various modules, and the system as a whole, function properly - i.e., the software executes, and executes correctly, as far as can be determined under

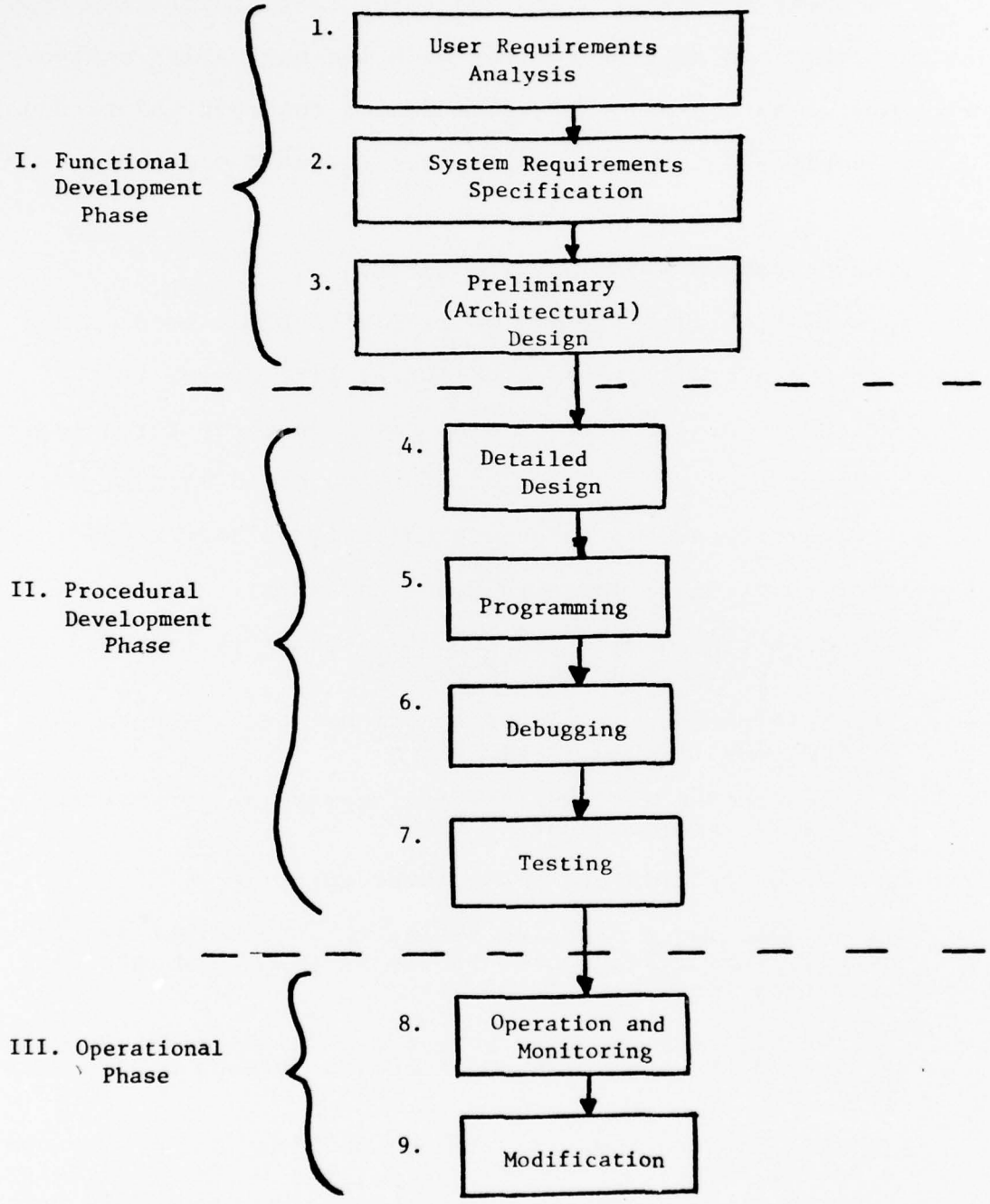


Figure 2.1

Phases of the System Life Cycle

test conditions (stages 6 and 7).

Finally, the third, or operational, phase involves:

(a) putting the system "on the air" - running it under real, as opposed to test, conditions in the user's environment, and monitoring its operation;

(b) making modifications, either to fix errors or make requested changes (stage 9).

The activities within the functional development phase are generally less well structured, the objectives and quality measures less well understood, than those in the procedural development or operational phases. Nor have the difficulties in the functional development phase gone unnoticed. A few studies have been undertaken to determine how well these tasks are usually executed. For example, Bell and Thayer (Bell and Thayer 1976) studied the software requirements specification problem in the context of both a large and a small system. They found a surprisingly large number of errors in this stage of the sample projects they examined - well over one error per page of requirements documentation (the small project gave rise to over 40 pages of such documentation, the larger project over 2500 pages).

Often, the difficulties within the functional development phase begin even earlier, in the user requirements analysis (sometimes referred to as "management information requirements analysis," or "MIRA") stage. In a recent survey, Carter (Carter 1975) asked a sample of professional systems analysts, and users, what they felt were the most critical factors contributing to successful development of information systems. Over 120 of these respondents claimed that "the correct identification of

management information needs" was the most critical such factor.

What makes system functional development so difficult? One key reason concerns the limited cognitive capabilities of the human brain. It has been widely observed that most people are able to deal conceptually with only seven or so distinct items of information at a time (Miller 1956). During functional development, the target system must be dealt with as a whole, and typically involves hundreds or thousands of variables, which effectively swamping any single designer's ability to keep mental track of all the pieces(1).

2.2 Application of Computers in Functional Development.

To help them cope with the complexity they face during functional development, some system designers have turned to a tool they know well: the computer. Various computer-based approaches have been developed for assisting designers in specifying and tracking system requirements, and in designing software. Some of these schemes are oriented toward specific kinds of computer systems: for instance, the SREM (System Requirements Engineering Methodology) approach for developing real-time command-and-control oriented systems ((Davis 1977), (Alford, 1977)). Others, such as PSL/PSA (Problem Statement Language/Problem Statement Analyzer), under development at the

---

(1) During later stages of the development cycle, the complexity problem is less severe, as any one person is required to deal with a relatively small "chunk" of the total system (e.g., program modules, or subroutines).

University of Michigan, are more general-purpose in their objectives.

The SDM methodology, with which this report is primarily concerned, is also a computer-based system aimed at supporting the functional development activity. Its orientation is primarily toward architectural design structuring. An overview of the SDM objectives and approach is given in the following section.

2.3 The SDM Approach.

The SDM approach to system design centers on the problem of identifying a system's modules, or "sub-problems", their functions, and their interconnections. It begins with a set of functional requirement statements for the proposed system, which are carefully edited so as to exhibit certain characteristics - implementation independence, unifunctionality, design relevance, etc. - necessary for such statements. Then each pair of requirements is examined in turn, by the system designer, and he makes a decision as to whether a significant degree of interdependence between the two requirements exists. This he determines by considering how each of the requirements might be implemented in the target system, then asking himself whether he envisions substantial interaction (either interference, or support) between them in the course of performing the implementation. Thus, while the requirements statements themselves are intended to be free of "implementation bias", the assessment of interdependencies demands consideration of

alternative modes of implementation. Then the resulting information is represented as a non-directed graph structure: nodes are requirement statements, links are assessed interdependencies. The graph is then partitioned, according to one of various possible algorithms, with the objective of locating a good decomposition. An index of partition goodness is employed, which incorporates measures of subgraph "strength" and "coupling." The actual goodness index is taken as the algebraic difference between the strengths of all the subgraphs, and the inter-subgraph couplings. That is, M = S - C, where S is the sum of the strength measures of all subgraphs, and C is the sum of all the inter-subgraph couplings.

Once an agreeable partition is determined, the resulting sets of requirements are interpreted by the designer as "design sub-problems." In essence, these design sub-problems, together with their interconnections (also derived directly from the graph partition) constitute the preliminary design. The overall SDM procedure is illustrated graphically in Figure 2.2.

Andreu (Andreu 1978) found, in applying this approach to a real set of requirements (for the design of a database management system), that iteration on the overall procedure was of value. The first pass through the SDM process produced a reasonable design, but this design was improved considerably in terms of clarity and completeness by studying it for weaknesses (typically, sub-problems with an unclear or complex functional interpretation, or cases of omitted specifications), then "completing" the set of requirements to fill identified gaps and

remove ambiguities.

This report is concerned especially with the transition from user needs to statements of system functional requirements (statements which may then be used as "input" to the SDM procedure - see Figure 2.2). Since a variety of schemes, developed over the last few years, purport to address the requirements statement problem, it is appropriate at this time to examine these approaches more closely. To that end, we turn now to an examination of one well-known, and reasonably typical, approach to specifying system requirements, the PSL/PSA system.

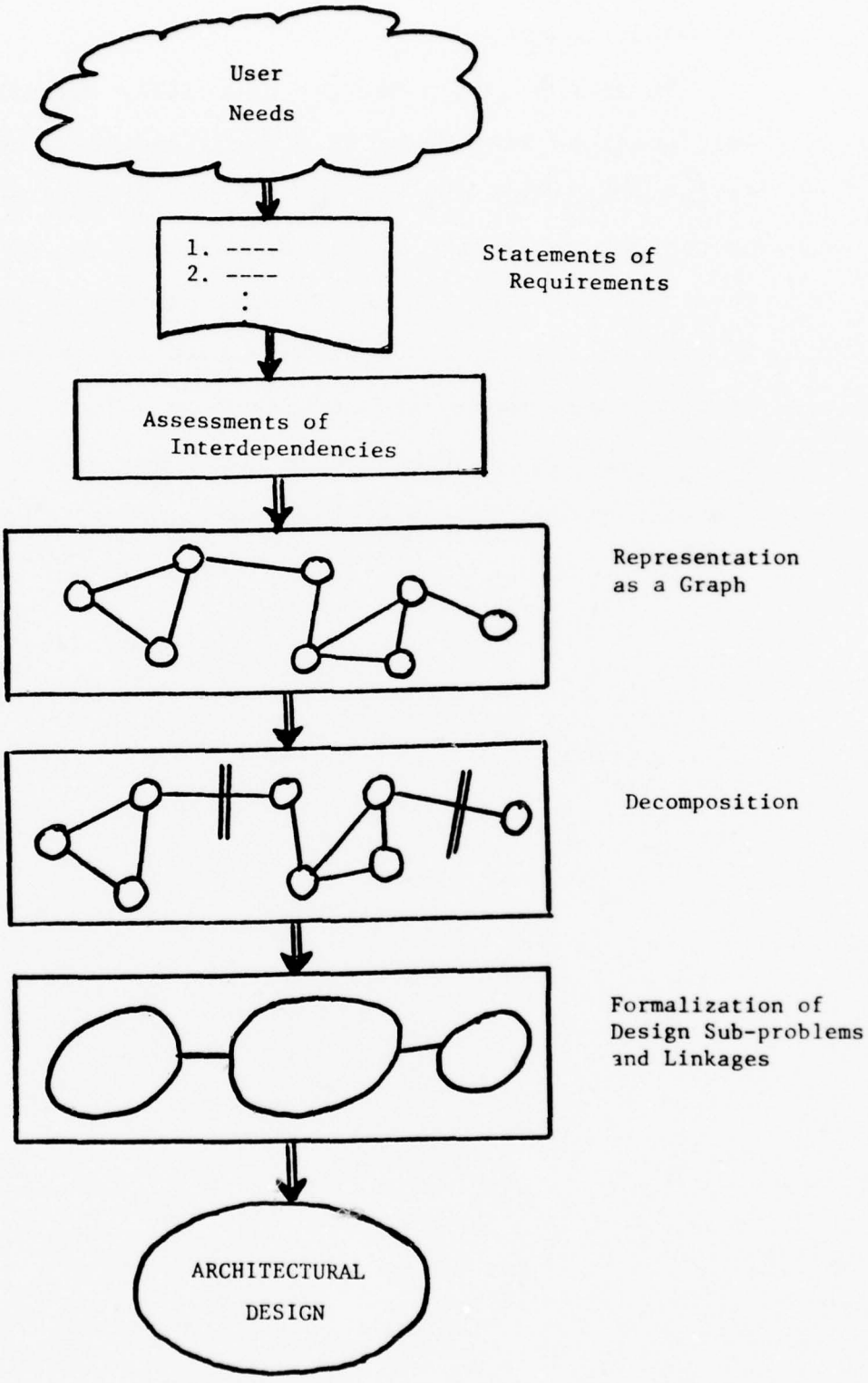


Figure 2.2

The SDM Design Methodology

3. Requirements' Statement Languages - The Case of PSL.

Recent years have witnessed a number of attempts to apply computers to the problems inherent in designing and building software systems. One relatively well-known approach is the ISDOS (Information System Design Optimization System) project, which was begun at the University of Michigan in 1969, and is ongoing (Teichroew 1971). Out of this project has come the PSL/PSA system, consisting of a formal language - PSL, or "Problem Statement Language" - for specifying a system's functional requirements, and a software support package - PSA, or "Problem Statement Analyzer" - for performing certain kinds of analysis on a set of machine-readable PSL statements, generating various reports from a set of such statements, etc. Our discussion here will focus almost entirely on PSL. For further information regarding the nature and capabilities of both PSL and PSA, see (Teichroew and Bastarache 1977).

3.1 The Structure of PSL.

The basic model underlying PSL is quite simple. PSL recognises two kinds of things: objects, and relationships. Objects are "things", such as data elements, logical collections of data, processes, etc. Each PSL object is given a unique name, and is classified as one of 22 possible object types (see Figure 3.1). These 22 object types may be grouped into four object classes: interface objects, target system objects, project management objects, and property objects. Interface objects (encompassing one object type) are used to describe the interface

Classes of Object Types	Object Type within Class
Interface	INTERFACE
Target System	
Collections of Information	INPUT,OUTPUT,ENTITY
Collections of Instances	SET
Relationships among collections of Information	RELATION
Data Definition	GROUP
Data Derivation	PROCESS
Size and Volume	INTERVAL,SYSTEM-PARAMETER
Dynamic Behavior	EVENT,CONDITION
Project Management	PROBLEM-DEFINER,MAILBOX
Properties	SYNONYM,KEYWORD,ATTRIBUTE, ATTRIBUTE-VALUE,MEMO,SOURCE,SECURITY

Figure 3.1

PSL Object Classes and Types

between the target system and its environment. Target system objects (12 object types) together with property objects (7 object types) are used to describe the target system. Project management objects (2 object types) are used to help document the organizational and project control aspects of the system development process.

The other top-level concept in PSL is that of relationship. Relationships are used to state ways in which PSL objects are

related to each other. PSL relationships may be likened to "verbs" which, together with PSL objects, serve to generate "sentences", or PSL statements. There are 58 different relationship types included in PSL, although many of these are "inverse pairs" (e.g., the pair RECEIVES, and RECEIVED BY). If we count each such pair as a single "distinct" relationship, the total number is reduced to 31. The various PSL relationships and complementary relationships are listed in Figure 3.2.

PSL semantics requires that only specific types of relationships may be used to interconnect any given pair of objects. For example, the object types ENTITY and PROCESS may legally be interconnected by certain relationships (e.g., process USES entity, or process UPDATES entity), but there are no legal PSL relationships that may interconnect object types ENTITY and INPUT. For a full discussion of PSL semantics, see (Teichroew and Bastarache 1977).

As PSL object types are grouped into object classes, so are relationship types grouped into classes of relationships. Eight relationship classes are defined in PSL, as a function of system "aspect". These eight classes are:

- system flow
- system structure
- data structure
- data derivation
- system size
- system dynamics

Relationship	Associated Complementary Relationship
ASSOCIATED	ASSOCIATED-DATA
ATTRIBUTES	--
BECOMING	WHEN
CARDINALITY	--
CONNECTIVITY	--
CONTAINED	CONSISTS
DERIVED	DERIVES
GENERATED	GENERATES
HAPPENS	--
IDENTIFIED	IDENTIFIES
INCEPTION	INCEPTION-CAUSES
KEYWORD	APPLIES
MAILBOX	APPLIES
MAINTAINED	MAINTAINS
PART	SUBPARTS
RECEIVED	RECEIVES
RELATED	BETWEEN
RESPONSIBLE-INTERFACE	RESPONSIBLE
RESPONSIBLE-PROBLEM-DEFINER	RESPONSIBLE
SECURITY	APPLIES
SEE-MEMO	APPLIES
SOURCE	APPLIES
SUBSET	SUBSETS
SUBSETTING-CRITERIA	SUBSETTING-CRITERION
SYNONYM	DESIGNATE
TERMINATION	TERMINATION-CAUSES
TRIGGERED	TRIGGERS
UPDATED	UPDATES
USED	USES
UTILIZED	UTILIZES
VALUES	--

Figure 3.2

PSL Relationships and Complementary Relationships.

- project management
- system properties.

PSL objects may also be re-classified according to this scheme. Figure 3.3 shows the classification of both objects and relationships according to system aspect. In cases of complementary relationship pairs, only one is shown.

Information which is needed to describe an object, and which cannot be specified using one or more relationships, can be included in a narrative description called a "comment entry." A number of different types of comment entries may be defined, depending on the type of object to which they pertain. These comment entries are shown starred in Figure 3.3. There are also certain other comment entries that may be used in every category; these entries are not shown in the figure.

3.2 Using Objects and Relationships to Create PSL Statements.

PSL statements have the general form

<object name> <relationship> <object name(s)>.

Typical examples are:

(a) payroll-process RECEIVES employee-work-data.

Here, payroll-process would be the name of a PROCESS object, and employee-work-data an INTERFACE object. RECEIVES is a system flow relationship. Note that an equivalent statement would be

employee-work-data IS RECEIVED BY payroll-process.

These two statements express complementary relationships, and are equivalent, both logically and semantically, within PSL.

(b) payroll-process SUBPARTS ARE payroll-data-read, pay-calculation, check-print.

This statement describes a hierarchical structure of processes. The right-hand side of the relationship consists of a list of three PROCESS objects. SUBPARTS ARE is a system structure relationship. Note that PSL can only describe hierarchically organized structures of data or objects.

(c) net-pay VALUES ARE Ø THRU 2ØØØ.

This example illustrates a somewhat different type of statement. Here, net-pay is an ELEMENT object (elementary data type), and the "relationship" is an expression of a range of values that net-pay may legally assume.

If we avoid double-counting statement types such as RECEIVES and IS RECEIVED BY as in the first example above, and omit system property and project management statements (these provide additional detail in a problem statement, but are in the nature of comment entries, as they are not analyzed by PSA), then there are 39 different PSL statement types. These are given in Appendix A in full detail, grouped according to system aspect, and are summarized in Figure 3.3. The system property and

project management statement types (see Appendix A) add another 11 to the list of statement types given in Figure 3.3. However, as these types serve only to add peripheral commentary to a PSL description, they are not strictly necessary in stating the logic of a processing problem.

PSL OBJECT TYPES AND RELATIONSHIP TYPES CLASSIFIED ACCORDING TO SYSTEM ASPECT.

System Aspect	Object Types	Distinct Statement Types	No. Statement Types
System Flow	PROCESS INTERFACE INPUT OUTPUT	RECEIVES, GENERATES RESPONSIBLE FOR	3
System Structure	PROCESS INTERFACE INPUT,OUTPUT SET,ENTITY ELEMENT,GROUP INTERVAL	PART OF CONTAINED IN SUBSET OF UTILIZED BY CONSISTS OF SUBSETTING-CRITERIA ARE	6
Data Structure	INPUT,OUTPUT SET,ENTITY GROUP,ELEMENT RELATION	IDENTIFIED BY RELATED TO...VIA CONSISTS OF CONTAINED IN BETWEEN...AND ASSOCIATED WITH	6
Data Derivation	PROCESS INPUT,OUTPUT SET,ENTITY ELEMENT,GROUP RELATION	USED BY USED BY...TO DERIVE USED BY...TO UPDATE DERIVED BY DERIVED BY...USING UPDATED BY UPDATED BY...USING MAINTAINS,PROCEDURE DERIVATION	10
System Size	PROCESS,EVENT SET,ENTITY ELEMENT RELATION INTERVAL SYSTEM-PARAMETER	VOLATILITY CONNECTIVITY IS...TO VALUE IS VALUES ARE...THROUGH HAPPENS...TIMES-PER	9
System Dynamics	PROCESS,EVENT SET,ENTITY INTERVAL CONDITION SYSTEM-PARA-METER	VOLATILITY VOLATILITY-SET HAPPENS...TIMES-PER VOLATILITY-MEMBER TRIGGERED BY INCEPTION CAUSES TERMINATION CAUSES TRUE/FALSE WHILE BECOMING TRUE/FALSE CALLED	9

Figure 3.3

3.3 An Illustration of PSL - Part A.

Using PSL involves four main activities: (a) identifying and naming objects, and assigning a unique type to each; (b) determining the relationships among the objects; (c) writing PSL statements to describe the target system; (d) writing comment entries to express any necessary information which cannot be expressed in the formal syntax.

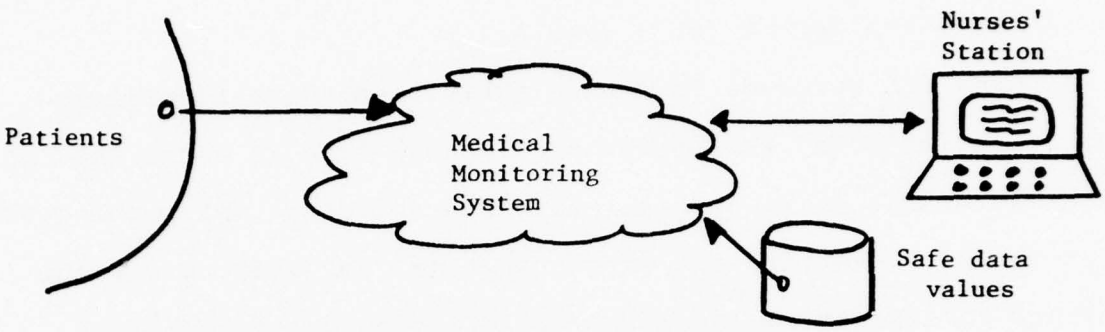
In order to illustrate both the general idea of PSL, and some of the detailed syntax, we consider here a specific example. The following brief description, slightly modified, of a Medical Monitoring System (MMS) was taken from Stevens (Stevens 1974):

> A patient medical-monitoring system is required for a hospital. Each patient is monitored by an analog device which measures factors such as temperature, blood pressure, pulse rate, and so on. The program reads these factors on a periodic basis. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range might be 98 to 99.5 degrees F.). If a factor falls outside a patient's safe range, the nurses' station is notified.

Clearly, the above is incomplete as a specification for such a system, but it is adequate for illustration purposes.

First of all, note that the MMS is a real-time system - when it is running, it interacts on a continuous basis with the patients being monitored. Also, the system presumably maintains a database of "safe" values, one set of values for each patient. We will assume a maximum of 10 patients, and round-robin monitoring, one patient's readings being taken every second.

The MMS structure may be sketched in a simple diagram as shown in Figure 3.4(a). The first-level PSL description of the MMS is given in Figure 3.4(b). The key elements of this



(a)

Diagram for a Simple Medical Monitoring System

FIRST-LEVEL PSL DESCRIPTION FOR THE MMS

```
INTERFACE patients;
     GENERATES patient-data;

INTERFACE nurses;
     RECEIVES help-signal;

INPUT patient-data;
     GENERATED BY patients;
     RECEIVED BY monitor;
     HAPPENS read-freq TIMES-PER minute;
     USED BY monitor TO DERIVE help-signal;

OUTPUT help-signal;
     GENERATED BY monitor;
     RECEIVED BY nurses;
     DERIVED BY monitor USING patient-data, safe-values;

SET safe-values;
     RESPONSIBLE-INTERFACE nurses;
     USED BY monitor TO DERIVE help-signal;

PROCESS monitor;
     RECEIVES patient-data;
     GENERATED help-signal;
     USES safe-values;

INTERVAL minute;

DEFINE
     read-freq SYSTEM-PARAMETER;
```

(b)

Figure 3.4

description include: (a) the identification of the INTERFACES that connect the MMS to the external environment; (b) the identification of the types of INPUT and OUTPUT data used or generated by MMS, and where obtained or used; (c) identification of a data SET, or collection of instances of data elements - the set of safe values; and, (d) the description of the PROCESS itself - the MMS analysis routines - and the inputs and outputs associated with it.

3.4 An Illustration of PSL - Part B.

We can expand and improve upon our earlier PSL description of the medical monitoring system by adding detail, describing more structure, etc. A more detailed depiction of the MMS might be as shown in Figure 3.5.

The MMS is shown there as consisting of three modules: one to monitor patient vital signs and compare to safe values, one to send an emergency message to the nurses' station in the event of monitored values exceeding the safe levels, and one to allow the nurses to modify the database of safe values.

A new PSL description of the MMS, containing more detail than the first description, is given in Figure 3.6. The changes from the first description include: (a) the addition of the EVENT object types named alarm and noalarm, and the CONDITION object type named safe-values-exceeded; (b) the hierarchical decomposition of the MMS into three sub-processes; (c) the hierarchical decomposition of the INPUT, OUTPUT, and ENTITY data into components (GROUPS and ELEMENTS); and (d) the

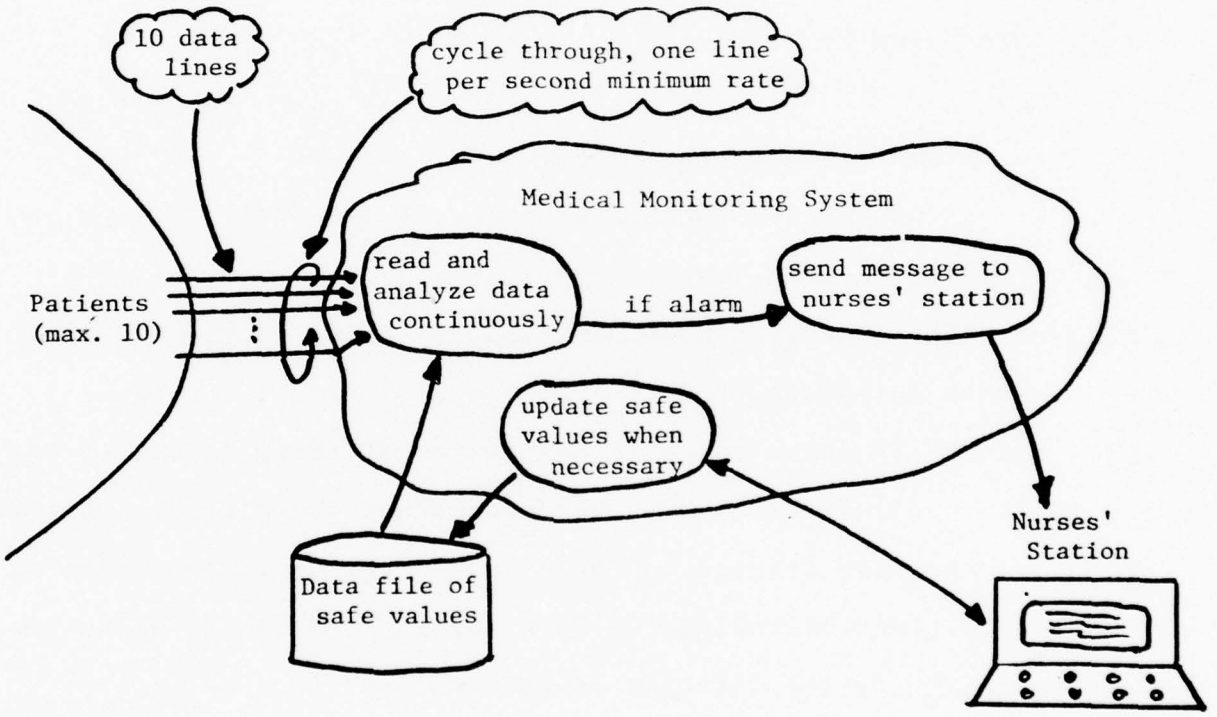


Figure 3.5

A More Detailed Medical Monitoring System

SECOND-LEVEL DESCRIPTION OF THE MMS

```
INTERFACE patients;
     GENERATES patient-data;
     DESCRIPTION;  between one and ten patients are
     are to be monitored in real time.
     Max. delay between readings on any patient
     is to be ten seconds;

INTERFACE nurses;
     GENERATES new-safe-values;

INPUT patient-data;
     GENERATED BY patients;
     RECEIVED BY analyze-vital-signs;
     CONSISTS OF patient-id-number, patient-vital-signs;
     HAPPENS read-freq TIMES-PER minute;

INPUT new-safe-values-info;
     GENERATED BY nurses;
     RECEIVED BY change-safe-values;
     CONSISTS OF patient-id-info, new-safe-values;
     USED BY change-safe-values TO UPDATE safe-values;

OUTPUT help-signal;
     GENERATED BY send-alarm-to-nurses;
     RECEIVED BY nurses;
     CONSISTS OF patient-id-info, patient-vital-signs;

SET safe-values;
     RESPONSIBLE-INTERFACE IS nurses;
     USED BY analyze-vital-signs TO DERIVE help-signal;
     UPDATED BY change-safe-values USING
     new-safe-values-info;
     CONSISTS OF safe-value-records;
     CARDINALITY IS 10;
     SECURITY IS authorized-nurses;

ENTITY safe-value-records;
     CONTAINED IN safe-values;
     CONSISTS OF patient-id-info, safe-values-info;
     IDENTIFIED BY patient-id-number;
     USED BY analyze-vital-signs;
     UPDATED BY change-safe-values;
     CARDINALITY IS 10;
     VOLATILITY;
            Any safe-value-records entity may be modified,
            deleted, or added to the data base by
            authorized nurses at any time. This will most
            commonly occur when the patient list changes;
```

```
GROUP patient-id-info;
     CONSISTS OF patient-name, patient-id-number;
GROUP safe-values-info;
     CONSISTS OF safe-temp, safe-bloodp;
GROUP patient-name;
     CONSISTS OF patient-first-name, patient-last-name;
GROUP safe-temp;
     CONSISTS OF safe-temp-low,safe-temp-high;
GROUP safe-bloodp;
     CONSISTS OF safe-bloodp-disatolic-low,
     safe-bloodp-diastolic-high,safe-bloodp-systolic-low,
     safe-bloodp-systolic-high;

ELEMENT patient-first-name, patient-last-name;
ELEMENT patient-id-number;
     IDENTIFIES safe-value-records;
ELEMENT safe-temp-low, safe-temp-high;
ELEMENT safe-bloodp-diastolic-low,
     safe-bloodp-diastolic-high,safe-bloodp-systolic-low,
     safe-bloodp-systolic-high;

PROCESS monitor;
     SUBPARTS ARE analyze-vital-signs,
     send-alarm-to-nurses, change-safe-values;

PROCESS analyze-vital-signs;
     RECEIVES patient-data;
     USES safe-values;

PROCESS send-alarm-to-nurses;
     USES patient-data, safe-values;
     GENERATES help-signal;
     TRIGGERED BY alarm;

PROCESS change-safe-values;
     RECEIVES new-safe-values-info;
     USES new-safe-values-info TO UPDATE safe-values;
     PROCEDURE;
             An authorized nurse updates the data base of
             safe values whenever an old patient is
             disconnected from the system, or a new patient
             connected.  Also, current safe values may be
             altered as patients' conditions require;

EVENT alarm;
     WHEN safe-range-exceeded BECOMES TRUE;
     TRIGGERS send-alarm-to-nurses;

EVENT noalarm;
     WHEN safe-range-exceeded BECOMES FALSE;
```

```
CONDITION safe-range-exceeded;
     BECOMING TRUE CALLED alarm;
     BECOMING FALSE CALLED noalarm;
     TRUE WHILE;
             Any vital sign for any patient connected to the
             MMS falls outside the defined safe range;
     FALSE WHILE;
        Vital signs within safe range;

INTERVAL minute;

DEFINE read-freq SYSTEM-PARAMETER;
```

Figure 3.6

representation of the maintenance of the SET of safe values.

3.5 PSL As a Design Tool.

An important question that deserves comment at this point is, to what extent does PSL serve as a designer's decision support system, i.e., play the role of a design aid? An important early motivation for the development of PSL/PSA was to help automate the task of system design, as opposed to documentation (Teichroew and Sayari 1971). However, experience with PSL and other similar tools has seen them used primarily as documentation techniques, not design techniques (Teichroew and Hershey 1977).

For example, in the MMS design illustrated in Figure 3.5 above, the designer (the author) decided, intuitively, that a three-module decomposition of the monitoring routine would be appropriate. The only justification for picking these three modules, with these functional characteristics, was that they "seemed reasonable", based largely on previous general software

design experience. Once the function of each module, and the data acted upon or interchanged among them, had been mentally worked out, PSL was then employed effectively to formally describe the scheme. However, PSL itself did not directly aid the designer in deciding on the system's structure, or on the functions of the components.

It is reasonable to infer, then, that specification (or "problem statement") languages like PSL, while they may be effective tools for gathering together and documenting information important to system design, in general do not fulfill the role of a methodology for guiding or assisting designers in conceiving system architectures. In contrast, the SDM approach is specifically oriented toward design assistance, and only secondarily towards documentation. This observation raises the question whether there might be a fortuitous combination of SDM and a scheme such as PSL/PSA that would effectively address both design support and documentation. Such a possibility is a subject for additional research, and will be addressed in a later report.

Research and practice in the field of requirements specification and preliminary system design has given rise to a number of important new concepts. However, it has also generated considerable confusion over interpretation of these concepts, much of which is due simply to a confusion over terminology. The purpose of the next section is to attempt to clarify some of the issues and problems in the requirement specification and system

design fields. We will attempt to shed some light on certain heavily-used, but vaguely-defined terms, and to relate this present research - its objectives, methods, and concepts - to other work in the area, including application of PSL/PSA, in the context of a simple framework to be presented there.

4. Terminology and Concepts: Some Clarifications.

The literature that focuses on the functional development phase of CSDD efforts exhibits much variation in content. There seem to be few unambiguous reference points - researchers, authors, and system designers have not yet agreed on precise terminology to describe what they do. This lack of agreement, understandably, adds confusion to both research and practice in this area.

In this section, we examine certain key terminological and conceptual topics, attempt to remove some of the ambiguity surrounding them, then consolidate them in a simple framework. This framework will prove useful in thinking about requirements specification and CSDD activities in general, and for relating the present research activity with other projects in this area.

4.1 Levels of Procedurality.

A widely used, but frequently misunderstood concept is that of procedurality. Programming languages are frequently described as being either "procedural" or "non-procedural" in nature. The usual definition centers around the distinction between stating or describing what is to be done (non-procedural) as opposed to how it is to be accomplished (i.e., the "procedure" to be followed, hence procedural). The main problem with this distinction is that it is put forth as a black-versus-white characterization. In practice, no clear dividing line exists between the two alternatives; rather, they should be viewed as the ends of a continuum, thus:

procedural <----------> non-procedural

There are, then, different levels of procedurality - different degrees to which a statement (in particular, a requirement statement) exhibits either a "what to" or a "how to" nature. Furthermore, whether a particular statement, command, etc., is viewed as procedural or non-procedural depends on the viewpoint of the person involved. The following example should help clarify this distinction.

Consider an assembly language programmer, involved in writing the code for a calculation module which is to become the DCF (Discounted Cash Flow) subroutine for a financial analysis package. Faced with the task, say, of adding together two values, he would think in terms of instructions such as the following:

```
L     1,A     (Load "A" into register 1)
A     1,B     (Add "B" to contents of register 1)
ST    1,C     (Store results of addition in "C")
```

Now, conventionally, this set of instructions would be described as procedural in nature. The programmer has to know, for example, that the "procedure" for adding two numbers together involves loading a register with the first number, adding the second into the register, then storing the sum. Such considerations as loading and storing the registers, not to mention the use of assembler mnemonic codes and instruction

format, are viewed as "how to", or procedural, aspects of performing this task.

In contrast, suppose the programmer were to make use of some high-level language, such as Fortran or PL/1. The addition operation might then be coded as (using PL/1)

```
C = A + B;
```

Now, from the assembly language programmer's point of view, such an encoding of the task is non-procedural: he no longer needs to be concerned about loading or storing registers, or about the other procedural aspects of the task. He can express the "what" aspect directly(1).

This is the point at which the standard characterization of the distinction between procedural and non-procedural would end. However, suppose we go one step further, and examine the same task from the point of view of, say, an eventual user of the financial analysis package (a non-programmer). As far as he is concerned, a typical non-procedural command might simply be

```
DCF PROJECT_X
```

which would execute the DCF module upon a given set of data,

---

(1) Of course, someone or something must worry about register loading, etc. In this case, the burden is shifted to the PL/1 compiler, or, if you prefer, the compiler designer.

PROJECT_X. This command states, at his level of concern, what is to be done. The fact that, at some point within the DCF module, two values had to be added together using the PL/1 statement C=A+B;, is a procedural issue concerned with how the DCF calculation is to be carried out. So that which was viewed as non-procedural by the assembly language programmer, is clearly a procedural issue as far as the end user is concerned.

To summarize, then, the procedurality level of a formal language, command, etc., is not absolute, but rather must be related to the viewpoint of the user of that language, and the nature of the decision problem upon which he is working.

An important reason for presenting this detail here is that problem statement languages (e.g., PSL) are often characterized as being "non-procedural". The appropriateness of such a characterization depends very much on the user's point of view. If the user is a system documentor, such a characterization may be appropriate; for a user who is trying to design a system, the characterization may be quite inappropriate. We will elaborate this distinction further at the end of this section.

4.2 Types of Requirements.

The literature in the systems design field also exhibits considerable ambiguity about what is meant by the "requirements" for a software system. Part of the reason for this is that the notion of requirements is very general. The term itself is used in many different contexts, and as a result these different contexts start becoming blurred in the mind of the reader, as

well as designer. For example, systems analysts refer to "information requirements analysis" as well as "system requirements specification"; to "functional requirements" as well as "design requirements"; and to "user requirements" as well as "software requirements specification". There is a growing number of methodologies (discussed in more detail shortly) that purport to address the problem of "requirements specification", whatever that may be defined to be in any particular case. Examination of some of these methodologies indicates that, not infrequiently, one person's "requirements" turn out to be another person's "detailed system design".

Now, at a high enough level, the concept of a system's requirements seems quite clear: these are statements of what the system is to do. Unfortunately, this definition is too general to be of much use. For example, "the system should help me control my inventory" is a requirement statement, as is "the file selection module must verify that file names are no longer than six characters." Clearly, these statements (a) are at different levels of abstraction, and (b) exhibit different degrees of procedurality.

In fact, it is insightful to classify requirements statements along each of these dimensions. This idea is pursued further in Section 4.4, in the context of the framework described there. One objective of that framework is to more precisely clarify the meaning of "requirements" especially with respect to

the system development cycle.

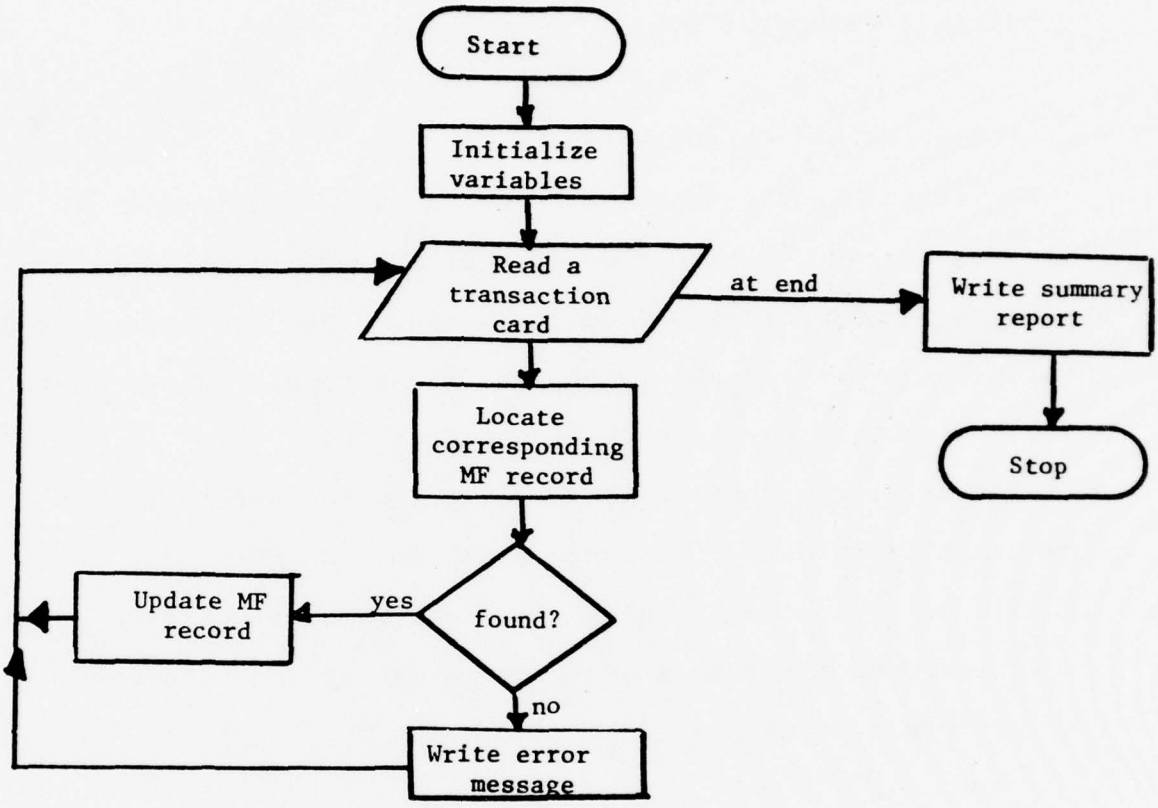
4.3 Processes and Capabilities.

As system requirements move toward (a) lower levels of abstraction, and (b) higher degrees of procedurality, they are also altered along another key dimension: they are transformed from statements of capabilities (which the target system is to possess) into statements regarding processes.

The difference between capability-type requirement statements (CS's) and process-type statements (PS's) may be highlighted with some examples. Typical CS's (taken from a set of requirements for a data base management system, discussed in greater detail in the next section) might be:

"Inter-file relationships can be described at run time";
"The maximum size of a field is at least 100 characters";
"The system will have a report break control feature".

In contrast to these are process-oriented statements. A relatively detailed example of a PS is a common program flowchart, for example, Figure 4.2(a). Many other forms, at varying levels of detail, may also be found. Another example would be I.B.M.'s HIPO diagrams, and SofTech's SADT charts; both are graphical approaches to specifying PS's. The Requirements Statement Language (RSL), developed by the Ballistic Missle Defense Advanced Technology Group (Alford 1977) and the Problem Statement Language (PSL) discussed earlier, are typical formal



(a)

```
INPUT patient-data;
     GENERATED BY patients;
     RECEIVED BY monitor;
     HAPPENS read-freq TIMES-PER minute;
     USED BY monitor TO DERIVE help-signal;

OUTPUT help-signal;
     GENERATED BY monitor;
     RECEIVED BY nurses;
     DERIVED BY monitor USING patient-data,
     safe-values;

PROCESS monitor;
     RECEIVES patient-data;
     GENERATES help-signal;
     USES safe-values;
```

(b)

Figure 4.2

Examples of Process-oriented Requirement Statements

language mechanisms for stating process-oriented requirements. For comparison purposes, an example of the latter, duplicated from an earlier example, is shown in Figure 4.2(b).

4.4 A Framework for Requirements Statements.

We have examined three important aspects of requirements statements: degree of procedurality, level of abstraction, and capability-versus-process. These characteristics may be viewed together to form a simple framework for thinking about and describing requirements in the context of the system development life cycle. The suggested framework is portrayed schematically in Figure 4.3, below. In the figure, requirements statements have been classified along two dimensions: level of abstraction, and degree of procedurality. The activities often referred to as "management information requirements analysis" (Taggart 1977) fall into the upper left quadrant of the diagram. Of course, not all systems development efforts are aimed at providing information to managers, but usually some user clientele is identifiable, and activities to elicit their needs, at a relatively high logical level, generally initiate the system development process.

As the process progresses, requirements are made more specific, often through some sort of hierarchical decomposition. Attempts are made to identify errors, omissions, conflicts, and other such problems with the requirements. Iterations are common. Often, problems encountered with "lower level" requirements necessitate alterations at "higher" (closer to the

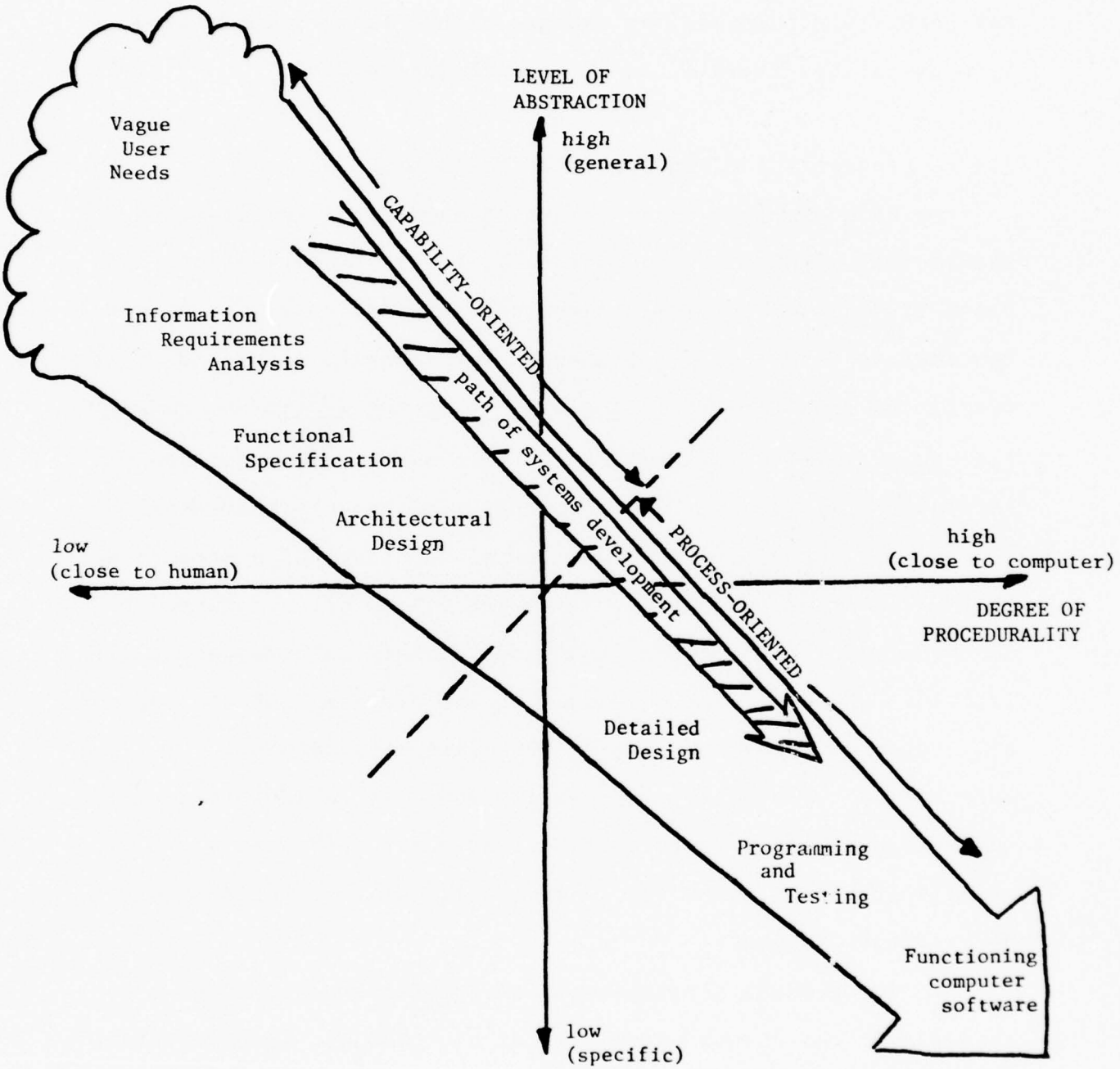


Figure 4.3

A Framework for Requirement Statements

user) levels. In fact, it has been argued that additional time spent in the early stages of system development is usually well invested. Boehm, for example, contends that each additional unit of resource (principally user and system analyst time) expended during the early stages returns between 1.5 and 3 units later on (during coding, testing, and modification stages) (Boehm, 1974).

As the development process enters the detailed design stage, requirements are usually translated into statements of what activities or data transformations will take place within specific elementary program modules, or subroutines. Control flow descriptions and interface specifications become important at this stage. While these statements are still usually termed "requirements", it is clear that their nature is significantly different from those requirements hammered out between users and analysts at the beginning of the design process. The transformation that the original requirements undergo in the course of completing the detailed design is precisely the embodiment of the software design and development process itself.

Eventually (if all goes well), requirements end up as computer programs. Of course, in traversing the path from user needs to PL/1 statements, considerable iteration is usually required, most often to clarify ambiguous or inconsistent requirements, or to fill in missing ones. It has been observed (Bell 1976) that, even under the most stringent conditions (e.g., designing the systems to support the Apollo moon launchings), it is effectively impossible to make an original set of requirements specifications complete. The necessity for iterations should not

be viewed as an aberration of what ideally ought to be a linear activity; rather, it is an inherent, important part of the overall system design process.

A somewhat different way of characterizing the CSDD activities is in terms of capability statements and process statements. Initial conceptual work, groping by users and analysts as to the nature of the decisions requiring support, information needs, and so forth, usually lead to capability-type requirement statements ("the system must be capable of supporting up to five users simultaneously, "the system must support both on-line and batch access", etc.). As the design activity proceeds, process-type requirement statements usually emerge. These PS's serve to guide and document detailed design and programming activities to follow.

The transition from CS's to PS's to coded software is not (necessarily) very "neat", however. In many CSDD cases, the CS's are never formally stated at all; they occur by default, by designer "assumption" (which are frequently at odds with users' "assumptions"), or by necessity (because other requirements constrain the design in various ways, often without explicit recognition). Also, it is not always all that clear whether a particular requirement statement is a CS or a PS. For example, the requirement that "the system should be able to print the output records in both sorted and unsorted form" specifies both a capability (being able to sort a file) and a process (sorting).

The distinction between CS's and PS's is important for the following reason: essentially all the currently available tools

and techniques for aiding and guiding the CSDD activities are process-type techniques. Included in this group are:

- HIPO (IBM)
- TAG (IBM)
- SOP (IBM)
- ADS (Honeywell, and ISDOS)
- SADT (SofTech Inc.)
- PSL/PSA (ISDOS)
- SREM and RSL (TRW Inc. and DMDATC)
- HOS (Draper Labs)
- HDM (SRI).

Consider again, for instance, the case of PSL. PSL is usually referred to as a non-procedural language for defining and describing a software system. However, keeping in mind that procedurality is really a continuum, a descriptive scheme such as PSL appears quite non-procedural as compared, say, to the assembly code eventually used to implement the system, but at the same time appears rather procedural as compared to the original English description of what the system was supposed to do. In particular, the PSL statements serve to structure an initial functional description of a system into (1) a set of data descriptions, (2) a set of processing descriptions, and (3) a set of other, miscellaneous items (EVENTS, CONDITIONS, INTERVALS, and the like). PSL, then, presents a relatively process-oriented description, consequently falls somewhere in the lower right

quadrant of the diagram in Figure 4.3.

This is an important point, and warrants further elaboration. Conventionally, PSL is referred to as a language for stating software functional specifications. However, the point being made here, by way of example, is that

"if a factor falls outside the patient's safe range, the nurses' station is to be notified"

is a good example of a true "functional" specification (one that is rather general in scope). However, the PSL statements

```
PROCESS send-alarm-to-nurses;
     USES patient-data, safe-values;
     GENERATES help-signal;
     TRIGGERED BY alarm;
```

form part of the description as to how the previous function is to be carried out: a procedure (subroutine?) named "send-alarm-to-nurses" will be executed under certain conditions ("alarm"), will read the current data from the monitoring lines, will read the corresponding safe values from the patient's data file, and will send an appropriately formatted message to the terminal at the nurses' station. The PSL description is much more procedural and process-oriented than the earlier functional specification from which it was derived.

We can generalize the foregoing argument to essentially all well-known "functional specification stating" tools, languages, media, etc. References to these schemes, including the ones listed above, may be had through various reviews, such as (Cougar 1973),(Teichroew 1972), and (Burns 1974). While the various

techniques differ substantially in detail, the above comments regarding PSL apply in all the cases we have examined: that these schemes generally are positioned toward the process-oriented, procedural end of the spectrum presented in Figure 4.3; that they tend to be design documentation (or program documentation) techniques as opposed to design decision support systems; that they come into play after the architectural design of the system has been created, not before or during and (this is the really confusing part) that they are frequently referred to, both by their creators, and sometimes even their users, as methodologies for functional specification, or problem statement, or system design.

The argument at this point is not that there is anything wrong with these PS methodologies per se, but that there are important steps in the system design process that must be taken - either planned or not, either supported or not - before process-oriented tools such as these can be brought into play. Central among these precedent activities is architectural design. Designers who seek to employ the process-oriented design aiding tools too soon in the development life cycle are, in effect, institutionalizing the tendancy to underplay, skip over, or at the worst totally ignore that part of the functional development phase which addresses capability-type requirement specifications. Our research in requirements definition and software architectural design is centrally concerned with this problem.

5. Expressing Functional Requirements.

It would be useful to pause for a moment at this point to review some of the issues that have been raised, and points that have been made so far.

The focus of the SDM research is the problem of architectural design of software systems. The general approach developed by Andreu involves the application of partitioning algorithms to a graph representation of a design problem, and the interpretation of the resulting clusters and linkages as design sub-problems. Since in this report we are concerned especially with the problem of expressing functional requirements for a target system (to be used as "input" to the SDM procedures), and since there is a number of well-known "requirements specification" systems currently available, in Section 3 we examined a representative system: PSL/PSA, developed out of the ISDOS project. We then turned to an examination of certain terminological and philosophical issues that tend to cloud discussions of system design and software engineering. In the light of this examination, we argued that PSL, while possessing its own strengths and weaknesses, was in fact more oriented towards documentation than design, was more process-oriented than capability-oriented, and was more procedural than would be appropriate at the architectural design stage.

We concluded, then, that PSL, and by extension other methodologies exhibiting similar characteristics, are not very appropriate mechanisms for expressing functional requirements as a lead-in to system architectural design. An alternative

approach for addressing this problem, which we call the "template approach", is described in this section.

5.1 The Format of Typical Functional Specifications

The SDM methodology takes as input a set of functional requirement specifications for the target system, and as such is dependent on both the existence and the appropriateness of the specifications.

First of all, it is clearly necessary that a system's requirements be formally stated - i.e., written down - before SDM may be applied. Unfortunately, it is not at all uncommon for the requirements for proposed systems to never be committed to paper, especially in the case of smaller systems or systems being developed "in-house" (as opposed to contracted development). Nevertheless, for our purposes we will assume that this first step has been taken, that requirements have been generated in some written form.

Then, given that specifications have been formalized and written down, the second problem concerns the appropriateness of the format in which they have been stated. Three important characteristics of requirement statements to be used in the SDM methodology include:

(a) unifunctionality - each statement describes a single function (not multiple functions) to be featured in the target system;

(b) implementation independence - each statement should be implementation free, i.e., ought to specify what is required of the target system but not how that requirement is to be met;

(c) common conceptual level - all requirement statements

should be, to the extent possible, at the same level of generality, or abstraction(1).

In order to test out some of the earlier SDM concepts, Andreu managed to locate a set of requirement statements(2), for a database management system, that came "pre-packaged" in a format reasonably suitable for the analytical approach he had developed. As examples of these requirements, the following three are typical:

- "Inter-file relationships can be described at run time;"
- "The maximum number of interrelated files is at least ten;"
- "User can cancel active request without loss of data".

The full list of Andreu's DBMS requirements is given in AppendixB.

Even though these requirements were in a roughly appropriate format from the outset, Andreu found that they had to be examined rather closely, and a number of them had to be edited somewhat, in order that they possess the three characteristics outlined above.

Unfortunately, functional requirements, when they formally exist at all, generally are not expressed in this format. More typical is the paragraph describing the requirements for the Medical Monitoring System, discussed earlier (see page 25). As

---

(1) See (Andreu 1978) for a more complete discussion of the characteristics of "good" requirement statements, in the SDM context.

(2) The requirements used by Andreu were issued by a U. S. Government agency as part of a procurement procedure.

another example of "typical" requirement statements, consider the following specification for the file system portion of an operating system specified by Honeywell for the U. S. Navy's All Application Digital Computer (AADC).

> "The definitions of logical files are carried out by a collection of tasks directed towards file creation, file retention, file destruction, and file access actions. These tasks are accessible to other OS tasks and to application tasks. File definition tasks utilize the input/output tasks to manipulate and create various directory records (not PF, permanent file, directories) of files. Requests must be sent to the basic executive to initiate file definitions and to confirm access privledges to protected information. File definition tasks provide a user (system or application) with a mechanism for establishing logical files. The mapping function between logical and physical descriptors are established and the protection machinery invoked. Files will physically exist on devices such as drums or tapes. Since different physical devices usually exhibit differing physical capabilities, the logical file description will establish the logical capabilities of the file relative to the supplied device (Honeywell 1972)."

The problem is, then, that if the SDM methodology for architectural design is to be widely applicable, it will be necessary to have a means of "translating" such typical functional specification statements into an appropriate form - i.e., a form exhibiting the three characteristics discussed earlier. An initial step towards defining such a mapping is proposed here.

5.2 Requirement Statement Templates.

We would like to have a simple procedure by which we could map general English-prose style requirement statements to a format suitable for input into the SDM procedure. As a first step in this direction, it would be useful to be able to identify a set of requirement statement "types", or templates, that might

be used as a skeleton upon which specific sets of requirements could be constructed. Such a set of templates would

(a) help to guide the thinking of the analyst in setting up the system specification for SDM analysis;

(b) help to insure that the resulting statements met the appropriateness criteria outlined earlier; and

(c) form the basis for further study and research of the requirements specification process generally.

To determine such a set of specification templates, the DBMS requirement set employed by Andreu was studied in detail. While the set included over 100 original requirements, close examination indicated that many of the statements exhibited similar patterns. For example, the three statements

- "Variable-sized fields can be defined";
- "Record-level lockout capability";
- "Report break control feature supported"

all basically state specific features the target system is to possess.

Similarly, certain other patterns are discernible in the DBMS requirements. Specifically, upon thorough analysis, precisely six templates were distilled from the 100 different statements. These six templates are given in Figure 5.1.

REQUIREMENT STATEMENT TEMPLATES

A. Existence.

There (can/will) be <modifier> <object>

B. Property.

<Mod> <object> (can/will) be <mod> <property>

C. Treatment.

<Mod> <object> (can/will) be <mod> <treatment>

D. Timing.

<Mod> <object> (can/will) <timing relationship> <mod> <object

E. Volume.

<Mod> <object> (can/will) be <order statement> <index> <count

F. Relationship.

a. Subsetting.

<Mod> <object> (can/will) contain <mod> <object>

b. Independence.

<Mod> <object> (can/will) be independent of <mod> <object>

Figure 5.1

Specific terms used in the template descriptions in Figure 5.1 are defined below.

Objects. Objects are defined to be of two types: items, and activities. Examples of item objects are:

- (file) size
- (interactive query) facility
- (user) request.

Examples of activity objects include:

- (system) set-up
- (database) maintenance.

Modifiers. Modifiers are strings of English adjectives that serve to further describe the associated object. Examples of modifier strings are shown above, in parenthesis.

Properties. A property is a word that describes some particular feature of the associated object. Examples include

- self-documenting
- queryable
- distributed.

Treatments. Treatments are words that describe something that can be done to the associated object. Examples include

- saved
- sorted
- locked.

Timing Relationship. Pairs of activity objects may be temporally related via timing relationships. For example,

- occurs before
- triggers
- occurs during.

Order Statements. An order statement specifies an order relation (< , <= , = , >= , >) between an object and a measure (defined below). Typical order statements are

- less than
- no more than (i.e., less than or equal to)
- at least (i.e., greater than or equal to).

Measure. A measure consists of a parameter and a unit. The parameter may be either a constant or a variable, and the unit may be either a simple unit (e.g., hours, dollars) or a compound unit (e.g., man-months, or dollars per month). Examples of measures include

- 2 hours
- 95 percent
- M1 man-months
- 120 characters per second.

Imperatives. Each template may occur in either of two imperative forms, distinguished by the use of either "can" or "will". The use of "can" indicates that the target system is to be capable of supporting the requirement being described, but that in any

particular implementation that feature may or may not be so utilized. In contrast, the use of "will" indicates that the feature described in that requirement must be included as an imbedded part of the target system. Generally speaking, only one form ("can", or "will") makes sense in any given requirement statement. For examples illustrating the differences between the two forms, consider the following property statements:

- "data fields can be null", and
- "null fields will be identifiable".

The first statement indicates that it is possible (although not necessary) for data fields to be able to be set to a null value, whereas the second statement requires that null fields be identifiable (i.e., distinguishable from zero or any other valid value).

An example of each type of template is given in Figure 5.2. The full set of statements obtained by transforming Andreu's original DBMS requirements into template form is given in Appendix C.

EXAMPLES OF REQUIREMENT STATEMENT TEMPLATES

A. Existence.

There will be database-level security facilities.
(modifiers: database-level security; object: facilities)

B. Property.

System status will be queryable.
(mod: System; object: status; property: queryable)

C. Treatment.

Database can be initialized using system utility.
(object: Database; treatment: initialized; mod: using system utility)

D. Timing.

Schema validation will occur before database usage.
(mod: Schema; object: validation; timing relationship: occur before; mod: database; object: usage)

E. Volume.

Maximum recovery time will be no more than 24 hours.
(mod: Maximum recovery; object: time; order statement: no more than; measure: 24 hours)

F. Relationship.

(a) Subsetting.

Record selection criteria can contain boolean conditional expressions.
(mod: Record selection; object: criteria; rlnship: can contain; mod: boolean conditional; object: expressions)

(b) Independence.

Schema definition will be indep. of database usage.
mod object relnship mod object

Figure 5.2

5.3 Side Effects of Expressing Requirements in Template Form.

One of the interesting, and potentially important, results stemming from translating the DBMS requirements into template form is the fact that certain kinds of modifications had to be made to the original statements in order to perform the statement mapping. As a result, the "normalized" statements better met the appropriateness criteria for the SDM methodology, and the activity of normalizing the statements proved beneficial in reducing ambiguity and bringing about their clarification.

Four different effects were observed during the normalization process. First, and probably most important, by working the requirements statements into template form, one is forced to consider exactly what each statement means, and how it ought to be expressed in terms of the templates. With a little practice, statements or parts of statements that are ambiguous or unclear tend to stand out, as they tend to obstruct the transformation of the statement into a template form. As an example, statement 70 asserts that

"Application is transportable to/from Agency's existing systems."

In assessing which form this statement ought to be mapped into, a first step is to ask what the left-object is. In this case, "application" is the clear choice. However, the meaning of "application" is unclear: does it refer to application programs to be run on the target database system, or to some other application? Additional investigation showed that the reference was to application programs previously developed by the agency on an earlier DBMS, which they wished to be able to transfer later on, if necessary, to the new system for which the requirements had been issued. The statement is also a property-type statement of the "will" variety. The statement may be made clearer as follows:

Application programs will be transportable to/from
mod object property mod

the agency's other DBMS's.
mod(cont.)

In this case, as for most of the DBMS requirement statements used by Andreu, the statement was able to be mapped into template form rather easily, but in so doing, sufficient thought had to be given to the statement's meaning that imbedded ambiguities and other possible difficulties (the meaning of "application", here) could be exposed and corrected.

The second observed effect was that of having to break up a requirement into two or more pieces in order to map it into

template form. For instance, the original requirement 53 stated:

"Users can direct output to the system printer".

In fact, this requirement states two different things: there will be a "system printer capability" in the target system (this was not stated as a separate requirement elsewhere), and output may (optionally) be printed on the system printer. Therefore, this statement was split into an existence statement,

"There will be a system printer",
mod object

and a treatment statement,

"User output can be printed using system printer."
mod object treatment mod

One of the characteristics of good requirements statements, for the purposes of the SDM methodology, is that each statement capture a single functional requirement. Thus, the splitting of requirements, as demonstrated above, in order to map them onto templates, represents movement toward this goal.

A third issue brought to light by the requirements mapping activity concerns the necessity for inclusion of certain parts of some of the original statements. This is well illustrated by statement 77, which was originally

"Capability to support two or more concurrent queries in different stages of processing".

This statement was mapped into a volume template, to say in effect that the number of concurrent queries can be at least two. A question arises, however, over what is meant by the phrase "in different stages of processing", and why this is part of the requirement statement at all. It demands, for example, an understanding of what is meant by "stages of processing" in this context, an ambiguous notion at best. Also, it seems to have strong implementation overtones that would be desirable to avoid. What is really required is concurrent query processing capability (possibly with certain performance restrictions not mentioned here). One is lead, therefore, to either eliminate the last phrase from the specification, or else seek clarification from the user (in this case, the issuing agency). The final template form of this statement would then be

"Number of concurrent queries can be at least 2 units.
object mod order measure

As with the previous case, it might also be necessary to include an additional statement sprcifying the existence of a concurrent query capability.

The final type of issue raised by the statement mapping process concerns errors in the source statements. It is somewhat surprising, given the detailed examination these requirements have received already, that there would be any obvious errors remaining. However, consider statement 62:

"Average system recovery time is 2 hours

over a 30 day period".

In translating this statement using a volume template, the nature of the order statement ("is", here) was studied. Presumably, the intent of this requirement was that average recovery time be no more than 2 hours over a thirty-day period, not that it be exactly equal to 2 hours. The corrected statement was taken to be:

Average 30-day recovery time will be no more than 2 hours.

modifier — object — order statement — measure

5.4 Summary.

Some initial investigation indicates that the template approach to constructing functional requirement statements is an effective and useful way to prepare requirements for use in the SDM architectural design structuring methodology. An obvious next step in this work, one that is currently being studied, is to start with a functional requirements document in the more usual form - an English-prose description such as the two illustrated earlier - and attempt to apply this translation and structuring procedure to it. If the translation of "classical" functional specifications to template form can be accomplished in the general case, without great additional effort, the universality and utility of the SDM approach will be substantially enhanced.

6. Summary and Directions for Further Work.

The main focus of this paper has been the functional development phase of the system design life cycle (refer again to Figure 1.1). Initial background discussion motivated the need for research in this problem area. Section 2 introduced the SDM methodology for architectural design. The development and use of formal languages for stating system requirements was discussed in Section 3, and was illustrated with the PSL language. In Section 4, we attempted to identify and clarify certain key terminological and philosophical issues. In Section5, the problem of expressing functional requirements was discussed in general, and some ideas were presented for a method, based on statement templates, for mapping requirements from English prose to a form suitable for input into the SDM decomposition analysis.

As was pointed out in Section 1, there is little previous research - normative or descriptive - on the problems inherent in the functional development phase. In practical cases, the activities carried out during this phase are heavily the result of intuition, experience, judgment - and guesswork! But because what goes on, or ought to go on, in this phase has been researched so little, a very important aspect of such research must be trying to understand exactly what it is we are studying. The problems with terminology and concepts, discussed in Section4, are a manifestation of this "meta-problem".

If the SDM methodology is taken as a focal point, opportunities for future research may be described with respect

to it. These alternatives may be classified into five groups:

(a) structuring and stating requirements,

(b) determining requirements interrelationships,

(c) expanding the underlying graph model,

(d) expanding and improving the decomposition methods, and

(e) coupling the output of the architectural design activity together with the following life cycle stages.

We need to continue to improve our understanding of how to elicit requirements at the appropriate level of abstraction, and how to state them in a format most appropriate for input to the architectural design process. The template approach, outlined in the previous section, is a first step in this direction. Ideas for elaborating and testing the template approach were also outlined there.

Another activity that requires additional research is the assessment of interdependencies among requirements. This step is heavily dependent on designer knowledge and judgment (the SDM approach is, it should be kept in mind, a designer decision support system, not a design automation scheme). Nevertheless, there is a number of ways in which the assessment activity may be improved and made more efficient. Efficiency is a rather important concern, since the number of assessments that must be made grows as the square of the number of requirements. Preliminary experience of Andreu, Holden, and others with the assessment task has shown that one of the most important bottlenecks is the "simple" logistics of organizing and keeping track of the assessments. Consequently, computer-based tools to help the designer keep track of his assessments in a natural

textual form should help to improve the assessment process significantly. Also, more elaborate tools may be required to help designers classify and interrelate assessments according to various implementation issues, as would be required by an extension of the underlying graph model (discussed below).

The basic model underlying Andreu's original work is a simple undirected graph. In particular, links in the graph, representing requirement interrelationships, are taken as binary in nature: either an interrelationship exists (in the mind of the designer) or it doesn't. There are various ways in which the basic model might be extended to better capture design-relevant information. For example, weights might be assigned to links, to represent the "strength" of the interrelationship. Also, relationships between implementation issues might be represented and taken into account in the decomposition procedure. The current SDM graph decomposition methodology takes no account of these "second-order" relationships.

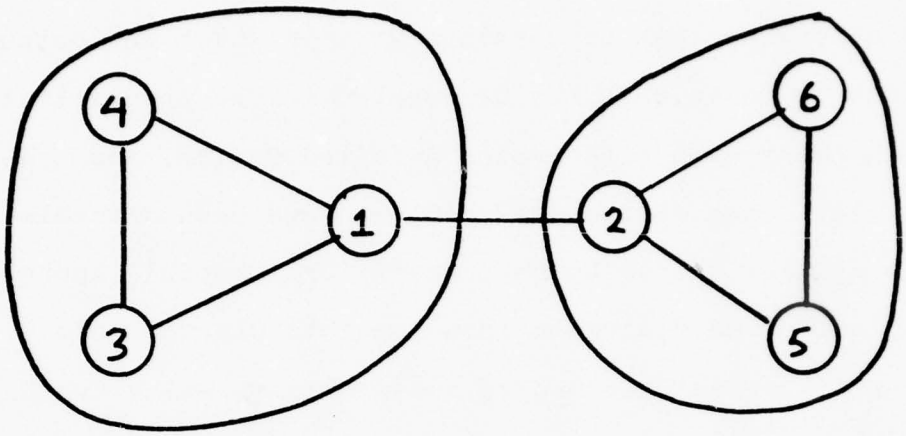
An example will help to illustrate the above points better. Figure 6.1(a) shows a six-node graph, frequently used for illustrative purposes by Andreu. The most natural partition of this graph - and the partition that also maximizes the goodness measure described earlier - is depicted in the figure.

In Figure 6.1(b), weight values, taken from a [0,1] range, with higher values representing a stronger interdependency (in the designer's eye), have been assessed for each link. Now it is less clear than before what the best decomposition ought to be. At any rate, the viability of decomposing the requirement set by

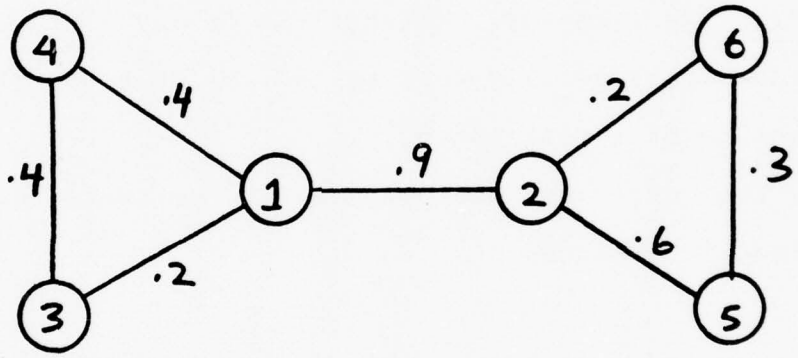
cutting through link (1,2) is clearly called into question, as this link has a considerably higher weight on it than on the other nearby links. This would indicate that the designer effectively believes there are strong (subjective) reasons for wanting to keep requirements 1 and 2 together in the same design sub-problem.

In Figure 6.1(c), two types of nodes are illustrated - requirement nodes, and implementation nodes. The dotted lines represent relationships between interdependencies. In the case represented by Figure 6.1(c), the intuitive argument for partitioning the graph in the same fashion as in the first case is again much less strong: to do so would isolate two design sub-problems that have a common type of implementation interdependency, which is probably not a good design decision.

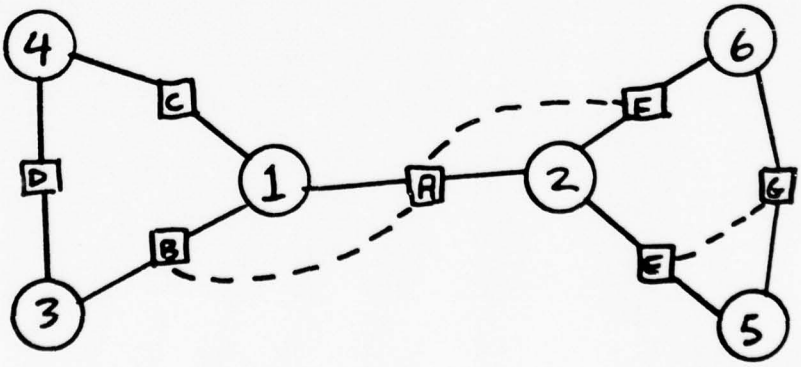
Of course, expanding the representation model is not a costless action. More complexity will be introduced into the model, necessitating additional effort on the part of the designer to map a real design problem onto the graph framework. Also, the graph decomposition and cluster analysis techniques developed by Andreu will need to be modified or completely re-worked. This represents yet another avenue for additional research. In this same direction, even if the underlying graph model is not expanded, there are a number of other wortnwhile investigations that might be undertaken with respect to the decomposition schemes. Generally, these issues are of second-order importance as compared to the others being discussed here.



(a)



(b)



(c)

Figure 6.1

Alternative Extensions to the Basic Graph Model

It is also important to consider ways in which the output of the SDM design techniques may be coupled to the stages that follow it in the system life cycle: detailed design, and implementation. Once design sub-problems have been determined and described, what should be done next? One possible approach is to bring a problem statement language into play at this point- i.e., to "write" the design sub-problems, which typically are at a lower level of abstraction, and hence in a form more amenable to procedural interpretation, in PSL or some other suitable descriptive language. Whether PSL or any of the other existing schemes is especially well suited for this use is another problem to be investigated.

Finally, the entire methodology needs to be tested out in various real-world settings.

REFERENCES

Alford, M.: "A Requirements Engineering Methodology for Real-time Processing Requirements", IEEE Trans. Soft. Eng., vol. 3, no. 1, 1977.

Andreu, R. C.: "Set Decomposition: Cluster Analysis and Graph Decomposition Techniques", Internal Report P010-01-01, M.I.T. Sloan School, September 1977.

Andreu, R. C.: "Solving Decomposition Problems: Alternative Techniques and Description of Supporting Tools", Internal Report P010-01-02, M.I.T. Sloan School, September 1977.

Andreu, R.C., and S. E. Madnick: "An Exercise in Software Architectural Design: From Requirements to Design Problem Structure", Internal Report P010-01-05, November 1977.

Andreu, R.C., and S. E. Madnick: "Completing the Requirements Set as a Means Towards Better Design Frameworks: A Follow-up Exercise in Architectural Design", Internal Report P010-01-06, December 1977.

Andreu, R. C.: A Systematic Approach to the Design and Structuring of Complex Software Systems, PhD. Thesis, Sloan School of Management, M.I.T., Cambridge, Mass. 1978.

Baker, F.: "Chief Programmer Team Management of Production Programming", IBM Systems Journal, vol. 11, no. 1, Jan. 1972.

Belford, P., et. al.: "Specifications: The Key to Effective Software Development", Proc. 2nd. Int. Conf. on Soft. Eng., 1976.

Bell, T., and T. Thayer: "Software Requirements: Are They Really a Problem?", Proc. 2nd Int. Conf. on Soft. Eng., 1976.

Boehm, B.: "Software and Its Impact: A Quantitative Assessment", Datamation, vol. 19, no. 5, May 1973.

Boehm, B.: "Some Steps Towards Formal and Automated Aids to Software Analysis and Design", Information Processing 74, North-Holland, 1974.

Brooks, P.: The Mythical Man-Month, Addison-Wesley, 1975.

Burns, I., et. al.: "Current Software Requirements Engineering Methodology", TRW Systems Group, Huntsville, Alabama, 1974.

Buston, J., and B. Randall: "Software Engineering Techniques", Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 1969.

Carter, D. M., et. al.: A Study of Critical Factors in MIS for

the U. S. Air Force, NTIS no. AD-A009-647/9WA, Colorado State University, 1975.

Cougar, J. D.: "Evolution of Business System Analysis Techniques", Computing Surveys, vol. 5, no. 3, Sept. 1973.

Davis, C., and C. Vick: "The Software Development System", IEEE Trans. Soft. Eng., vol. 3, no. 1, Jan. 1977.

DeWolf, B.: "Requirements Specification and Design for Real-time Systems: A Problem Statement", IR&D Memo No. 4, C. S. Draper Labs., Jan. 1977.

Holden, Timothy: "A Systematic Approach to Designing Complex Systems: Application to Software Operating Systems", Internal Report P010-7805-05, M.I.T. Center for Information Systems Research, May 1978.

Honeywell Inc.: Operating System/AADC: Preliminary Functional Specifications, Honeywell Systems and Research Division, Naval Air Development Center, Contract N62269-72-C-0051, 1972.

Miller, George: "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", Psychological Review, vol. 63, no. 2, March 1956.

Parnas, D.: "Information Distribution Aspects of Design Methodology", Information Processing 71, North-Holland 1971.

Parnas, D.: "On the Criteria to be Used in Decomposing Systems into Modules", Comm. of the ACM, vol. 15, no. 12, Dec. 1972.

Ross, D., and K. Schomann: "Structured Analysis for Requirements Definition", IEEE Trans. Soft. Eng., vol. 3, no. 1, Jan. 1977.

Stevens, J., et. al.: "Structured Design", IBM Systems Journal, vol. 13, no. 2, April 1974.

Taggart, W., and M. Tharp: "Survey of Information Requirements Analysis Techniques", Computing Surveys, vol. 9, no. 4, 1977.

Teichroew, D., and H. Sayari: "Automation of System Building", Datamation, vol. 17, no. 8, 1971.

Teichroew, D.: "A Survey of Languages for Stating Requirements for Computer-based Information Systems", Proc. of the Fall Joint Computer Conference, AFIPS Press, 1972.

Teichroew, D., and M. Bastarache: "PSL Users' Manual", ISDOS TR No. 98, March 1975.

Teichroew, D., and E. Hershey: "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Computer-based Information Systems", IEEE Trans. Soft. Eng., vol.

3, no. 1, Jan. 1977.

White, J., and T. Booth: "Towards an Engineering Approach to Software Design", Proc. 2nd Int. Conf. on Soft. Eng., 1976.

Weinberg, G.: The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.

APPENDIX A

Complete Listing of PSL Statement Types.

A.1 System Flow Statements.

(1) process/interface RECEIVES input/output

(1-a) input/output RECEIVED BY process/interface.

(2) interface RESPONSIBLE FOR set.

(2-a) set RESPONSIBLE-INTERFACE interface.

(3) process/interface GENERATES input/output

(3-a) input/output GENERATED BY process/interface.

A.2 System Structure statements.

(4) input/output/process/interface PART OF input/output/process/interface.

(4-a) input/output/process/interface SUBPARTS ARE input/output/process/interface.

(5) entity/input/output CONTAINED IN set/set/set

(6) set SUBSET OF set.

(6-a) set SUBSETS ARE set.

(7) set/set/set SUBSETTING-CRITERIA ARE subsetting-criteria/element/group.

(7-a) subsetting-criteria/element/group SUBSETTING-CRITERION set/set/set.

(8) process UTILIZED BY process.

(8-a) process UTILIZES process.

(9) interval CONSISTS OF interval (optionally preceded by

system parameter).

A.3 Data Structure.

(10) input/output/entity/set/group CONSISTS OF group or element/"/"/"/"

(11) entity IDENTIFIED BY group/element.

(11-a) group/element IDENTIFIES entity.

(12) entity RELATED TO entity VIA relation.

(12-a) relation BETWEEN entity AND entity.

(13) relation ASSOCIATED-DATA IS element/group.

(13-a) group/element ASSOCIATED WITH relation.

A.4 Data Derivation.

(14) input/entity/set/group/element USED BY process/"/"/"/"

(14-a) process/"/"/"/" USES input/entity/set/group/element.

(15) input/entity/set/group/element USED BY process TO DERIVE entity/set/element/group/output.

(15-a) process USES input/entity/set/group/element TO DERIVE output/entity/set/group/element.

(16) input/entity/set/group/element USED BY process TO UPDATE entity/set/group/element.

(16-a) process USES input/entity/set/group/element TO UPDATE entity/set/group/element

(17) output/entity/set/group/element DERIVED BY process.

(17-a) process DERIVES output/entity/set/group/element.

(18) output/entity/set/group/element DERIVED BY process USING input/entity/set/group/element.

(18-a) process DERIVES output/entity/set/group/element USING input/entity/set/group/element.

(19) entity/set/group/element UPDATED BY process.

(19-a) process UPDATES entity/set/group/element.

(20) entity/set/group/element UPDATED BY process USING input/entity/set/group/element.

(20-a) process UPDATES entity/set/group/element USING input/entity/set/group/element.

(21) set/relation DERIVATION comment-entry.

(22) relation/defined-name MAINTAINED BY process.

(22-a) process MAINTAINS relation, subsetting-criterion.

(23) process PROCEDURE comment-entry.

A.5 System Size.

(24) entity/set/relation CARDINALITY IS system-parameter.

(25) relation CONNECTIVITY IS system-parameter TO system-parameter.

(26) element/defined-name VALUE IS positive-integer.

(27) element/devined-name VALUES ARE minimum-value THRU maximum-value.

(28) process HAPPENS system-parameter TIMES-PER interval.

A.6 System Dynamics.

(29) input/output/event HAPPENS system-parameter TIMES-PER interval.

(30) entity VOLATILITY comment-entry.

(31) set VOLATILITY-SET comment-entry.

(32) set VOLATILITY-MEMBER comment-entry.

(33) process TRIGGERED BY event.

(33-a) event TRIGGERS process.

(34) process INCEPTION-CAUSES event.

(34-a) event ON-INCEPTION process.

(35) process TERMINATION-CAUSES event.

(35-a) event ON-TERMINATION process.

(36) condition BECOMING TRUE/FALSE CALLED event.

(36-a) event WHEN condition BECOMES TRUE/FALSE.

(37) condition TRUE/FALSE WHILE comment-entry.

A.7 Project Management.

(38) interface/input/output/entity/set/relation/ process/group/element/interval/condition/event/memo/defined-name RESPONSIBLE-PROBLEM-DEFINER person-name.

(38-a) person-name RESPONSIBLE FOR list as in (38).

(39) problem-definer-name MAILBOX mailbox-name.

A.8 System Properties.

Note: in the following, "list" refers to the list of object types used in (38) above.

(40) list SYNONYM synonym-name(s).

(41) list DESCRIPTION comment-entry.

(42) list (excluding "memo") SEE-MEMO memo-name.

(43) list KEYWORD keyword-names.

(44) list ATTRIBUTES attribute-name(s).

(45) list SECURITY security-name(s).

(46) list SOURCE source-name(s).

(47) list (excluding "memo", "defined-names") APPLIES name(s).

APPENDIX B

The Full Set of DBMS Requirements used by Andreu.

1. Data base schema (data dictionary) including inter-file relationships, is defined and maintained independently of database usage.
2. Separate files can be defined to be interrelated.
3. Data description language is English-like and self-documenting.
4. Database schema is validated by system prior to usage.
5. Interfile relationships can be defined at run time.
6. Field definition permits validation of input datum as to acceptable values.
7. The maximum number of files in the data base is at least 10.
8. Maximum number of interrelated files is at least 5.
9. Maximum size of a logical record is at least 500 information characters.
10. Maximum size of an item (field) is at least 100 characters.
11. Maximum number of items in a record is at least 50.
12. Repeated fields (multi-values attributes) can be defined.
13. Variable-sized fields can be defined.
14. Records in the database can be added.
15. Records in the database can be changed.
16. Records in the database can be deleted.
17. Database update can be performed by on-line user through query language.
18. Database maintenance can be performed by batch.
19. A bulk database update (initialization) can be performed through system utility.
20. Null-value field identification and generation supported.

21. Field update can trigger computation of a correlated field in same record.

22. Field update can trigger computation (e.g., tally of a correlated field in a different record).

23. Data integrity supported at least at file level lockout on update.

24. Record level lockout.

25. Checkpoint/restore facilities.

26. Transaction history facility.

27. Separate security facilities for retrieval and update.

28. Database level security.

29. File level security.

30. Field-level security.

31. Non-procedural user's language available for on-line query and update.

32. Boolean conditionals can be used in record-selection criteria.

33. Relational conditionals can be used in record selection criteria.

34. Arithmetic expressions can be used in record selection criteria.

35. Text-string scanning expressions can be used in record selection criteria.

36. Relational condition can compare variable to variable.

37. Data meeting selection criteria can be used for subsequent query processing.

38. Selected data can be stored by at least one sort key.

39. Capability for report formatting.

40. Report formatting optionally automatic.

41. Report break control feature supported.

42. Report summary line feature supported.

43. Multi-valued fields can be selectively listed.

44. Screen (menu) formatting facilities supported.

45. Local keyboard terminal supported.

46. Remote keyboard terminal supported.

47. CRT supported.

48. Delat Data 5260 supported.

49. User can interrogate status of system.

50. User can interrogate status of current report.

51. User can cancel active request without loss of data integrity.

52. User can suppress listing, save report, and later re-initiate listing.

53. User can direct output to system printer.

54. User can route listing to other terminal.

55. Capability to broadcast messages to all terminals.

56. Signon security.

57. A master terminal facility with privledged commands and control is supported.

58. Master terminal can be relocated to any on-line terminal.

59. System set-up effort, and each subsequent sysgen, less than one man-month.

60. Utilities to aid set-up.

61. Average up-time for minimum configuration at least 95 percent over a 30-day period.

62. Average system recovery time is 2 hours over a 30-day period.

63. Maximum recovery time is 24 hours.

64. Maintenance requirements less than 1 hour per week.

65. Power fail restart facility.

66. Dual processor fail-soft capability.

67. Removable disks containing data base can be mounted and processed during an on-line session.

68. A job accounting recording facility is supported suffiecent to charge users by application and by department.

69. A job accounting reporting facility.

70. Application is transportable to/from Agency's existing systems.

71. Data is transportable to/from Agency's existing systems.

72. System can operate in ordinary office environment.

73. System can communicate with other Agency's systems.

74. With a single terminal active, user can receive a response from a direct access to any item in the data base in less than 5 seconds.

75. Response time as independent as possible of file size.

76. Capability to support at least 10 active terminals.

77. Capability to support two or more concurrent queries in different stages of processing.

78. Display rate of terminal at least 120 characters per second on a CRT and 15 characters per second on a hard-copy terminal.

79. Dynamic file reorganization capability.

80. Dynamic reallocation of released and deleted storage areas.

81. File size limited only by disk storage capacities.

82. Total on-line data base can be distributed over many disk units.

83. Controls for tuning system performance at sysgen or system load time.

84. Controls for dynamically tuning system performance during run time.

85. Application tuning can be accomplished by restructuring the data to bias retrieval vs. update performance characteristics.

86. Can be delivered within M1 months.

87. Being used by at least 10 users within M2 months.

88. Non-recurring costs for basic configuration not more than C.

89. Maintenance costs less than MC per month.

APPENDIX C

Edited DBMS Requirements Transformed to Template Form.

The number in brackets refers to the requirement number used by Andreu (see Appendix B).

C.1 Existence Statements.

General template form:

There will be <mod> <object>.

1. (2) There will be file interrelationships.
2. (12) " Repeated field definitions.
3. (25) " checkpoint/restore facilities.
4. (26) " transaction history facilities.
5. (28) " database level security facilities.
6. (29) " file level security facilities.
7. (30) " field level security facilities.
8. (31) " non-procedural query/update language.
9. (39) " report formatting facility.
10. (41) " report break control facility.
11. (42) " report summary line facility.
12. (44) " menu formatting facility.
13. (45) " local keyboard terminal operation.
14. (46) " remote keyboard terminal operation.
15. (47) " CRT keyboard terminal operation.
16. (48) " Delta Data 5260 terminal operation.
17. (53-a) " system printer.
18. (55) " message broadcase facility.
19. (56) " signon security facility.
20. (57) " master terminal facility.

21. (57-b) " master terminal privledged commands.
22. (60) " set-up assistance utilities.
23. (65) " power fail restart capability.
24. (66) " dual-processor fail-soft capability.
25. (68-a) " job accounting recording facilities.
26. (69) " job accounting reporting facilities.
27. (73) " inter-system communication facilities.
28. (83-a) " static tuning controls.
29. (84-a) " dynamic tuning controls.

C.2 Property template statements.

General form for property template:

<mod> <object> *can/will be* <mod> <property>.

30. (3-a) Data definition language will be English-like.
31. (3-b) Data definition language will be self-documenting.
32. (20-a) Data fields can be null.
33. (20-b) Null fields will be identifiable.
34. (40) Report formatting facilities will be optionally automatic.
35. (49) System status will be queryable.
36. (50) Current request status will be queryable.
37. (67-a) Database disk units will be dismountable.
38. (70) Application programs will be transportable between existing systems.
39. (71) Data files will be transportable among existing systems.
40. (82) On-line databases can be distributed across multiple disks.
41. (85) Logical files can be alternately physically organizable.
42. (13) Data fields can be variable-sized.

C.3 Treatment template statements.

General form for treatment template is:

<mod> <object> can/will be <mod> <treatment>.

43. (6) Data items can be validated using field definition information.
44. (14) Database can be increased.
45. (15) Database records can be changed.
46. (16) Database records can be deleted.
47. (17-b) Database records can be updated using query language.
48. (18) Database records can be maintained using batch processing.
49. (19-a) Database can be initialized using system utility.
50. (19-b) Database can be updated using system utility.
51. (23) Files can be locked during update.
52. (24) Records can be locked.
53. (38-a) Selected data can be sorted.
54. (43) Multi-values fields can be listed selectively.
55. (51-a) Active requests can be cancelled.
56. (51-b) Data integrity will be maintained with respect to active request cancellation.
57. (52-a) Report listing can be suppressed.
58. (52-b) Suppressed report listing can be saved.
59. (52-c) Saved report listing can be re-initialized.
60. (53-b) Output can be printed using system printer.
61. (54) User-produced listing can be printed using alternative terminals.
62. (58) Master terminal can be relocated using alternative on-line terminals.
63. (79) Files can be re-organized dynamically.
64. (80) Released memory can be re-allocated dynamically.

C.4 Volume Template Statements.

General form for Volume Templates

<mod> <object> can/will be <order statement> <measure>.

65. (59-a) System set-up effort will be less than 1 man-month.
66. (59-b) System sysgen effort will be less than 1 man-month.
67. (61) Minimum configuration 30-day average up-time will be at least 95 %.
68. (62) Average 30-day recovery time will be no more than 2 hours.
69. (63) Maximum recovery time will be no more than 24 hours.
70. (64) Weekly maintenance time will be less than 1 hour.
71. (74) Single-active-terminal direct-access response time will be less than 5 seconds.
72. (76) Active terminal number can be at least 10 units.
73. (77) Concurrent query number can be at least 2 units.
74. (78-a) CRT display rate can be at least 120 characters per second.
75. (78-b) Hardcopy display rate can be at least 15 characters per second.
76. (81) File size can be equal to available physical storage size.
77. (86) Development time will be no more than M1 months.
78. (87) Ten-user installation time will be no more than M2 months.
79. (88) Basic configuration non-recurring costs will be no more than C dollars.
80. (89) Maintenance costs will be no more than MC dollars/month.
81. (7-b) Maximum number of database files can be at least 10 units.
82. (8-b) Maximum number of interrelated database files can be at least 5 units.
83. (10-b) Maximum number of field characters can be at least 100 units.
84. (9-b) Maximum number of logical record characters can be at least

500 units.

85. (11-b) Maximum number of logical record fields can be at least 50 units.

86. (38-b) Number of possible sort keys will be at least 1 unit.

C.5 Timing template statements.

General form for timing statement template is:

<mod> <object> can/will be <timing relationship> <mod> <object>.

87. (4) Schema validation will occur before database usage.

88. (5) Inter-file relationship usage can occur before database usage.

89. (17-a) Database update can occur before database on-line usage.

90. (21) Field update can trigger same-field-other-record update.

91. (22) Field update can trigger computation.

92. (67-b) Disk mounting can occur during on-line database usage.

93. (83-b) Static tuning can occur after database loading.

94. (83-c) Static tuning can occur after sysgen.

95. (84-b) Dynamic tuning can occur during database usage.

C.6 Relationship template statements.

General form for Relationship Subsetting template:

<mod> <object> can contain <mod> <object>.

96. (1-a) Schema definition can contain inter-file relationships.

97. (7-a) Database can contain files.

98. (8-a) Database can contain inter-related files.

99. (9-a) Logical records can contain characters.

100. (10-a) Fields can contain characters.

101. (11-a) Logical records can contain fields.

102. (32) Record selection criteria can contain boolean conditional expressions.

103. (33) Record selection criteria can contain relational conditional expressions.

104. (34) Record selection criteria can contain arithmetic expressions.

105. (35) Record selection criteria can contain text-string scanning expressions.

106. (36) Relational conditional expressions can contain variable-to-variable comparisons.

107. (68-b) Job accounting data will contain application charges.

108. (68-c) Job accounting data will contain organizational unit charge

General form for Relationship independence template:

<mod> <object> can/will be independent of <mod> <object>.

109. (75) Response time will be (as much as possible) independent of fi size.

110. (1-b) Schema definition will be independent of database usage.

111. (27) Retrieval security privileges will be independent of update security privileges.